Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Miha Pleško

Poenostavljen postopek za poganjanje aplikacije v okolju unikernel na oblačni platformi OpenStack

magistrsko delo

študijski program druge stopnje
računalništvo in informatika

Mentorica: prof. dr. Mojca Ciglarič

Ljubljana, 2016

Zahvala

Zahvaljujem se mentorici prof. dr. Mojci Ciglarič za pomoč pri uresničevanju ciljev naloge ter smernice pri pisanju samega dela. Zahvaljujem se uni. dipl. ing. Gregorju Bergincu za strokovno pomoč pri implementaciji nadgradnje ter dr. Justinu Činklju za svetovanje v primeru zapletov na nivoju unikernela. Hvala tudi staršem in Alji za podporo v času nastajanja magistrske naloge.

Miha Pleško, 2016

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
BR	Binary Translation	sopomenka za DRC
DNS	Domain Name System	sistem domenskih imen
DRC	Dynamic Recompilation	dinamično ponovno prevajanje binarne kode
GCC	GNU Compiler Collection	orodje GCC
IDE	Integrated Development Environment	pripomoček za razvoj programske opreme
IP	Internet Protocol	omrežni protokol
IPC	Inter-Process Communication	mehanizem deljenja podatkov med procesi
LAMP	Linux, Apache, MySQL, PHP	skupek programske opreme
NAT	Network Address Translation	način omrežnega naslavljanja
OS	Operating System	operacijski sistem
PCIe	PCI Express	standard za komunikacijo z napravami
RAMP	Rumprun, Apache, MySQL, PHP	skupek unikernelov za poganjanje spletne aplikacije
SELINUX	Security-Enhanced Linux	varnostni modul za Linux
SR-IOV	Single-Root I/O Virtualiz.	razširitev za PCIe
UTS	UNIX Timesharing System	mehanizem deljenja omrežnega imena med procesi
VM	Virtual Machine	virtualni stroj

Povzetek

Naslov: Poenostavljen postopek za poganjanje aplikacije v okolju unikernel na oblačni platformi OpenStack

Tehnologija unikernel se uveljavlja kot alternativa uporabi splošnonamenskega virtualnega stroja za poganjanje ene same aplikacije v računalniškem oblaku. Priprava unikernela je kompleksno opravilo, ki od uporabnika zahteva obvladovanje znanj, s katerimi se med razvojem aplikacije praviloma ne srečuje, kar vodi do frustracij. V magistrskem delu se postavimo v vlogo spletnega programerja in implementiramo preprosto spletno aplikacijo ter jo poženemo na oblačnem ogrodju OpenStack z uporabo tehnologije unikernel. Pri tem na podlagi lastnih metrik za merjenje uporabniške izkušnje identificiramo kritične pomanjkljivosti postopka in jih z nadgradnjo odprtokodnega orodja Capstan odpravimo. Nadgrajeno orodje, napisano v programskem jeziku Go, zniža kompleksnost priprave unikernela tako, da uvede javno dostopen repozitorij vnaprej pripravljenih modulov, iz katerih se pripravi končni unikernel. Delo sklenemo z demonstracijo postopka priprave unikernela z nadgrajenim orodjem, ki ga na podlagi predstavljenih metrik ocenimo kot uporabniku bolj prijaznega od izhodiščnega.

Ključne besede

unikernel, uporabniška izkušnja, računalniški oblak, OSv, Capstan, OpenStack

Abstract

Title: Simplified process for running application in unikernel environment on the OpenStack cloud platform

Unikernels are a new approach to operating system design that can be considered as a direct alternative to general-purpose virtual machines when it comes to running a single application. However, the unikernel contextualization process is complex and requires deep understanding of operating system design that application developers generally don't possess, resulting in frustrations. This thesis focuses on detecting and improving critical factors that have negative influence on user experience during the unikernel preparation process. We implement a simple web application and deploy it on OpenStack platform using unikernels to demonstrate the degree of process complexity. We then upgrade Capstan tool to significantly siplify the unikernel preparation process by introducing public repository of precompiled ready-to-use application packages. We conclude the thesis by demonstrating the improved user experience of the upgraded tool by deploying test web application on OpenStack.

Keywords

unikernel, user experience, cloud infrastructure, OSv, Capstan, OpenStack

Poglavje 1

Uvod

Unikernel je majhen sistemski posnetek, ki je ustvarjen z namenom poganjanja vnaprej izbrane aplikacije. Besedo **unikernel** v pričujočem delu uporabljamo kot prevzeto besedo, sposojenko, saj v času nastajanja dela še ne obstaja slovenska ustreznica. Prevedli bi jo lahko neposredno kot *enojedrnik*, vendar je beseda vsebinsko neustrezna. Boljši poskus slovenjenja bi bil opisni: *(eno)namenski sistemski posnetek*. Slednji izraz sicer bistveno bolje opiše samo tehnologijo, a je nepraktičen.

Na tehnologijo unikernel gledamo kot na popolnoma nov gradnik oblačnih infrastruktur. Unikerneli nam veliko obetajo s svojo prostorsko varčnostjo, hitrim zagonskim časom in ugodnimi pogoji s stališča varnosti [16]. Poganjanje aplikacije v unikernelu je alternativa poganjanju na splošnonamenskem virtualnem stroju. Slednji zaradi svoje splošnosti porablja razkošno količino dragocenih računskih virov in povečuje tveganje za vdor. Tehnologija unikernel se specializira v **poganjanje aplikacije na oblačni infrastrukturi** in skoraj v celoti odpravi odvečno kompleksnost in požrešnost splošnonamenskih virtualnih strojev. V unikernelu poganjamo izključno en proces, aplikacijo, brez odvečnih gonilnikov, brez storitev in brez pomožnih prednameščenih programov. Osvobodimo se vse odvečne teže, ki je z vidika poganjanja aplikacije nekoristna in le povečujejo tveganje za vdore.

Vendar pa tehnologija unikernel opisane prednosti prinese za določeno

ceno. Po eni strani obstajajo omejitve, ki narekujejo, kakšne aplikacije sploh lahko poganjamo v unikernelu. Če naša aplikacija ne ustreza omejitvam, jo moramo prilagoditi. Če je ne moremo prilagoditi, je ne moremo pognati na unikernelu in uporabiti moramo splošnonamenski virtualni stroj. Po drugi stani pa je unikernele potrebno pripraviti na poseben način. Za razliko od splošnonamenskih virtualnih strojev, ki vsebujejo prednameščen operacijski sistem z množico orodij, s katerimi namestimo svojo aplikacijo, unikernela ne moremo kar pognati, saj v njem še ni operacijskega sistema. Povedano drugače, unikernel moramo pripraviti (vanj zapeči našo aplikacijo) **preden ga lahko poženemo**. Aplikacija mora vsebovati tudi podmnožico funkcij operacijskega sistema (funkcije za upravljanje s pomnilnikom, funkcije za uporabo datotečnega sistema itd.)[1], zato moramo aplikacijo in operacijski sistem prevesti hkrati ter ju statično povezati med seboj[2]. Posledično na svojem razvojnem računalniku poleg orodij za prevajanje lastne aplikacije potrebujemo še vsa potrebna orodja za prevajanje funkcij operacijskega sistema iz izvorne kode ter znanje, kako jih uporabiti. Namesto aplikacije prevajamo torej dvoje, aplikacijo in operacijski sistem, kar tipično traja nekaj minut.

Zgrajeni unikernel ni nič drugega kot majhen sistemski posnetek, ki ga lahko poženemo na hipervizorju. Predstavljajmo si ga kot datoteko s končnico `.vdi`, kot jo poznamo iz programa VirtualBox [14] (točen format je seveda odvisen od implementacije unikernela). Z njim lahko počnemo vse, kar lahko počnemo s sistemskimi posnetki: lahko ga kloniramo, poženemo, zaustavimo, ponovno poženemo, lahko ga priklopimo v omrežje tipa NAT in posredujemo vrata (ang. port forwarding), lahko mu pripnemo novo napravo (ang. attach volume) in še kaj. Unikernel lahko brez posebnih težav poženemo na ogrodju OpenStack.

Opisane prednosti unikernelov so nas vzpodbudile, da smo pričeli iskati možnosti za izboljšavo njihove očitne pomanjkljivosti, kompleksnosti priprave. V magistrskem delu smo se torej usmerili v izboljšanje uporabniške

[1] Izbira podmnožice in njena implementacija je odvisna od implementacije unikernela.

[2] To ne velja le za redke implementacije unikernelov, na primer OSv, ki uporabljajo dinamični povezovalnik.

izkušnje pri izgradnji unikernelov. Naš namen je bil izogib frustracij, ki jih trenutno doživi programer aplikacije, ko jo želi pognati na ogrodju OpenStack z uporabo unikernlov.

Struktura dela je sledeča. V poglavju 3 se seznanimo z osnovnimi koncepti virtualizacije. Definiramo pojme kot so virtualni stroj, hipervizor in vsebnik, ki jih v razdelku 3.3 uporabimo za umestitev tehnologije unikernel. V poglavju 2 predstavimo metodologijo, ki smo jo uporabili pri izvedbi praktičnega dela magistrskga dela. Postavimo se v vlogo spletnega programerja in implementiramo preprosto spletno aplikacijo Medo, opisano v razdelku 4.1, in se jo namenimo pognati v lastni lokalni postavitvi oblačnega ogrodja OpenStack, opisani v razdelku 4.2. V razdelku 5.1 definiramo kriterije za ocenjevanje uporabniške izkušnje, ki jih v razdelku 5.2 uporabimo za oceno uporabniške izkušnje pri uporabi unikernelov tipa Rumprun in unikernelov tipa OSv. V poglavju 6 se lotimo nadgradnje obstoječega orodja za izgradnjo unikernelov tipa OSv, imenovanega Capstan. V poglavju 7 na kratko opišemo integracijo orodja Capstan v novo nastajajočo platformo za upravljanje z unikerneli, UniK. V poglavju 8 ovrednotimo uporabniško izkušnjo, ki jo nudi nadgrajeno orodje Capstan. Sklenemo s poglavjem 9, kjer predlagamo nekaj idej za nadaljnje delo.

Poglavje 2

Metodologija

Magistrsko delo se osredotoči na problematiko poganjanja ene same aplikacije v posamezni virtualizacijski enoti računalniškega oblaka. Opazimo namreč, da je vsakokratna uporaba splošnonamenskega virtualnega stroja potraten način za poganjanje aplikacije.

Prvi korak je **študija področja** (poglavje 3), kjer se seznanimo z obstoječimi tehnikami virtualizacije in oblačnim ogrodjem OpenStack. Na podlagi študije izberemo ustrezno virtualizacijsko enoto, unikernel, s katero lahko nadomestimo splošnonamenski virtualni stroj.

Drugi korak je **vzpostavitev testnega okolja** (poglavje 4). Pripravimo testno okolje z lokalno postavitvijo oblačnega ogrodja OpenStack in implementiramo testno spletno aplikacijo. V tem koraku na podlagi študije izberemo podmnožico implementacij tehnologije unikernel, ki izpolnjuje osnovne predpostavke testne aplikacije.

Tretji korak je **definicija lastnih kriterijev** za ocenjevanje uporabniške izkušnje pri uporabi dane implementacije tehnologije unikernel (razdelek 5.1). Kriterije definiramo tako, da čim bolj nazorno opišejo uporabniško izkušnjo z vidika končnega uporabnika unikernela, tj. spletnega razvijalca.

Četrti korak je **ovrednotenje uporabniške izkušnje** postopka uporabe unikernela za izbrano podmnožico implementacij tehnologije unikernel (razdelek 5.2). S pomočjo lastnih kriterijev ovrednotimo postopek ustvarjanja

unikernela za lastno testno aplikacijo in njegovega zaganjanja na lastnem testnem okolju. Na podlagi ocene se odločimo za eno izmed implementacij tehnologije unikernel, OSv, katere uporabniško izkušnjo v nadaljevanju izboljšamo.

Peti korak je **nadgradnja orodja** Capstan za pripravo unikernela OSv (poglavje 6). Z njo naslovimo pomanjkljivosti, ki smo jih odkrili med vrednotenjem uporabniške izkušnje. Kakovost nadgradnje zagotovimo s pomočjo medsebojnega pregledovanja programske kode.

Šesti korak je **ponovno ovrednotenje uporabniške izkušnje** postopka uporabe unikernela OSv (poglavje 8). Pri tem uporabimo iste pogoje in kriterije za ocenjevanje uporabniške izkušnje, kot smo jih uporabili pred nadgradnjo.

V zadnjem koraku podamo **sklep** (poglavje 9), kjer tudi izpostavimo možnosti za nadaljnje delo.

Predstavljene metodologije smo se tekom izdelave magistrskega dela držali z eno izjemo. Po uspešni implementaciji nadgradnje orodja Capstan se je na portalu GitHub nepričakovano pojavila odprtokodna platforma UniK, ki naslavlja podoben problem, kot naša nadgradnja. Odzvali smo se s študijo omenjene platforme, ki ji je sledila integracija našega nadgrajenega orodja v platformo. Zavoljo popolnosti magistrskega dela v poglavju 7 opišemo platformo UniK in način omenjene integracije.

Poglavje 3

Pregled področja

Poganjanje dveh ali več izoliranih okolij na istem fizičnem računalniku označujemo s pojmom *virtualizacija* in je osnova vseh oblačnih infrastruktur. Na virtualizaciji temeljijo tako javne oblačne infrastrukture, na primer Amazon Web Services [20], Google Compute Engine [15] in Microsoft Azure [31], kot tudi zasebni oblaki, ki jih upravljamo z orodji kot sta OpenStack [27] in Docker [19].

Najpogostejši sta dve tehniki virtualizacije [10]: virtualizacija s hipervizorjem in virtualizacija z vsebniki. V nadaljevanju pregleda področja najprej predstavimo posamezno tehniko virtualizacije. Pridobljeno znanje nato uporabimo za umestitev tehnologije unikernel v poglavju 3.3. Pregled področja sklenemo s kratko predstavitvijo oblačnega ogrodja OpenStack, ki je priljubljeno orodje za upravljanje z virtualizacijskimi enotami in temelji predvsem na virtualizaciji s hipervizorjem.

3.1 Virtualizacija s hipervizorjem

Pri virtualizaciji s hipervizorjem sta pomembna dva pojma, ki sta tesno povezana med seboj: virtualni stroj in hipervizor. Popek et al. so v [24] že leta 1974 postavili naslednjo definicijo virtualnega stroja:

Definicija 1. *VIRTUALNI STROJ je okolje ustvarjeno s hipervizorjem,*

in naslednjo definicijo hipervizorja:

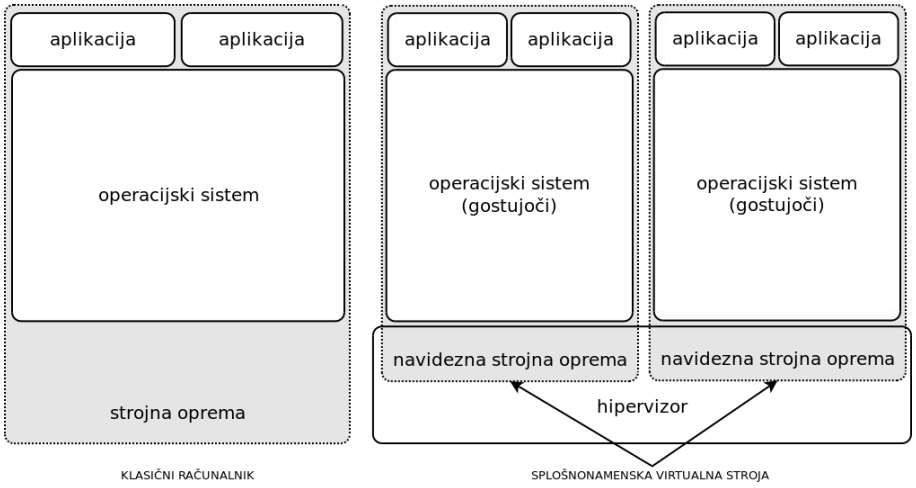
Definicija 2. *HIPERVIZOR je program, ki ustvarja okolje z naslednjimi tremi lastnostmi: (a) je skoraj identično kot okolje, ki ga nudi dana strojna oprema, (b) programi, ki tečejo v njem, se morajo v najslabšem primeru izvajati malenkost počasneje, kot bi se neposredno na dani strojni opremi, in (c) ima popoln nadzor nad sistemskimi viri okolja.*

Oglejmo si podrobneje, kaj pomenijo te lastnosti za hipervizorja. Predpostavimo, da se v virtualnem stroju izvaja gostujoči operacijski sistem, kot prikazuje slika 3.1.

(a) Lastnost **ekvivalence izvajanja** narekuje, da mora biti izvajanje gostujočega operacijskega sistema znotraj izoliranega virtualnega stroja popolnoma ekvivalentno, kot če bi ga izvajali na fizičnem stroju, z dvema izjemama. Prva izjema je *čas izvajanja*, ki je za isto zaporedje ukazov na virtualnem stroju lahko daljši kot na fizičnem stroju. Daljši časi nastopijo zaradi orkestracije deljenja istega fizičnega vira med več virtualnih. Druga izjema je *razpoložljivost virov* - znotraj virtualnega stroja se lahko zgodi, da je vir v danem trenutku zaseden, čeprav ga gostujoči operacijski sistem ne uporablja. Začasno mu ga namreč lahko zasede drug virtualni stroj. Predstavljajmo si na primer 2 GB fizičnega pomnilnika, ki si ga delita dva virtualna stroja, vsak z 1.5 GB virtualnega pomnilnika. Če prvi porabi vseh 1.5 GB, ki so mu na voljo, bo drugi lahko porabil 0.5 GB, nato bo pomnilnik v zasedenem stanju, čeprav na videz izgleda, kot da je še 1 GB razpoložljivega.

(b) Lastnost **učinkovitosti** narekuje, da se mora statistično gledano večina procesorskih ukazov gostujočega operacijskega sistema izvesti neposredno na fizičnem procesorju, torej brez vmešavanja hipervizorja. Po tej lastnosti hipervizor ločimo od emulatorja in interpreterja.

(c) Lastnost **nadzora nad viri** narekuje, da gostujoči operacijski sistem pod nobenim pogojem ne sme imeti možnosti uporabe virov, ki mu



Slika 3.1: Virtualni stroj je okolje, ki je skoraj identično okolju, ki ga nudi strojna oprema, le da so viri navidezni. Na sliki vidimo klasični računalnik (*levo*) brez virtualizacije. Aplikacije tečejo s pomočjo operacijskega sistema, ki teče neposredno na strojni opremi računalnika. Podobno je pri virtualnih strojih (*desno*), le da tam operacijski sistem nevede teče na navidezni strojni opremi. Pridevnik **gostujoči** poudarja dejstvo, da ta operacijski sistem teče s pomočjo gostitelja (hipervizorja), ki zanj pripravi navidezne vire.

niso eksplicitno dodeljeni. Hipervizor mu dodeli določene vire (disk, pomnilnik, mrežno kartico in druge) in gostujoči operacijski sistem ne more, na primer, dostopati do dela pomnilnika, ki mu ni dodeljen. Popoln nadzor nad sistemskimi viri pomeni tudi, da mora biti hipervizor zmožen dodeljene vire pridobiti nazaj.

3.1.1 Načini namestitve hipervizorja

Ločimo tri načine namestitve hipervizorja: kot aplikacija, kot sestavni del operacijskega sistema in kot samostojen operacijski sistem (glej sliko 3.2). Oglejmo si posamezne načine namestitve.

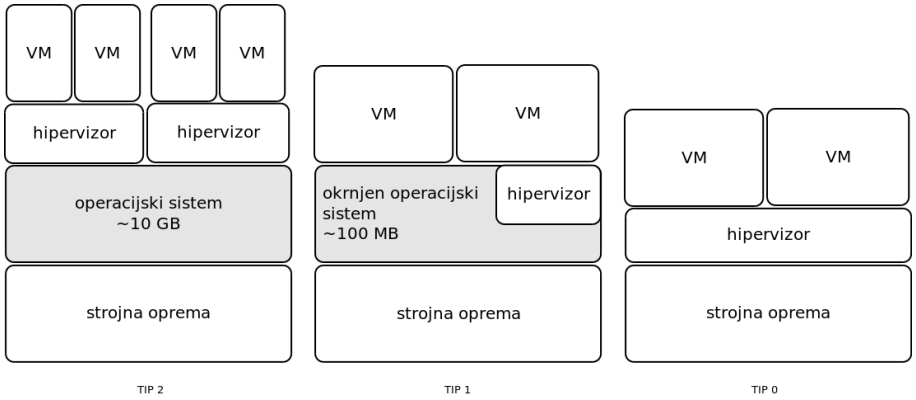
TIP 2: hipervizor kot aplikacija (ang. Type-2 Hypervisor)
Namestimo ga znotraj splošnonamenskega operacijskega sistema. Hipervizor ne implementira neposredne komunikacije s strojno opremo, saj namesto njega za to skrbi operacijski sistem. Takšen način uporabe hipervizorja je zaradi fleksibilnosti zelo razširjen, najbolj znane implementacije so VirtualBox, VMware, QEMU in Hyper-V. Kot vsaka druga aplikacija je takšen hipervizor izpostavljen težavam z učinkovitostjo, varnostjo in zanesljivostjo.

TIP 1: hipervizor integriran v operacijski sistem (ang. Type-1 Hypervisor)
Obstajajo posebni operacijski sistemi z vgrajenim hipervizorjem. Takšen hipervizor ima boljši nadzor nad komunikacijo z mehanskimi napravami, zato je zmogljivejši kot hipervizor tipa 2. Tudi težave z varnostjo in zanesljivostjo so pri takšni uporabi bistveno manjše, saj je upravljanje s hipervizorjem dovoljeno le sistemskemu administratorju. Kot primer omenimo Xen in VMware ESXi.

TIP 0: hipervizor kot operacijski sistem (ang. Type-0 Hypervisor)
Skrajni primer je namestitev hipervizorja neposredno na BIOS/UEFI tako, da prevzame vlogo minimalističnega operacijskega sistema, na katerem uporabniku ni dovoljeno nameščati lastnih aplikacij, ampak le upravljanje z virtualnimi stroji. Takšen tip hipervizorja je še v razvoju in potrebuje podporo v strojni opremi, zato v trenutku pisanja tega dela nismo našli še nobenega konkretnega primera implementacije.

3.1.2 Tehnike virtualizacije

Naloga hipervizorja je poganjanje izoliranih okolij na istem fizičnem stroju, virtualizacija. Poznamo dve osnovni tehniki virtualizacije: popolno virtualizacijo in paravirtualizacijo, kot prikazuje slika 3.3. Pri prvi tehniki lahko v virtualnem stroju uporabimo nespremenjen operacijski sistem, pri drugi pa mora biti operacijski sistem ustrezno prilagojen.

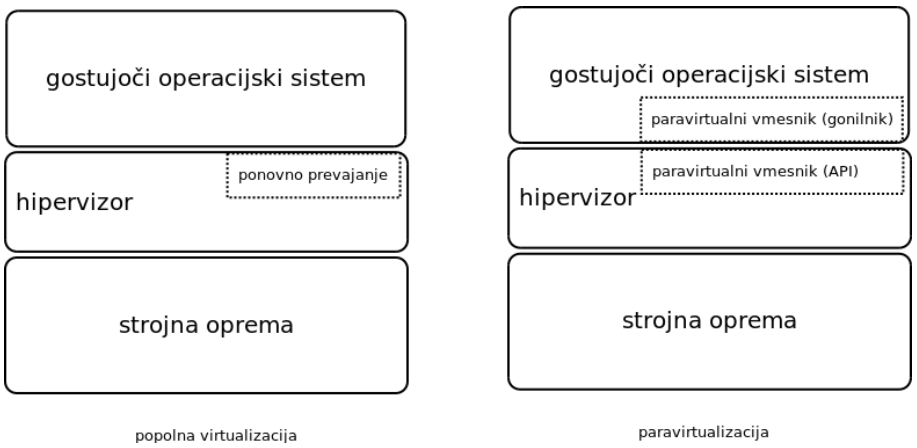


Slika 3.2: Trije tipi hipervizorjev: TIP 2 je zmogljivostno najšibkejši izmed treh, vendar je zaradi svoje splošnosti in kompatibilnosti najbolj priljubljen (VirtualBox, VMware). TIP 1 je zmogljivostno boljši kot TIP 2, vendar je potrebno stalno skrbeti za ustrezne gonilnike strojne opreme, zato se uporablja le v profesionalnih okoljih.

Popolna virtualizacija

Popolna virtualizacija (ang. full virtualization) je virtualizacija, kot si jo običajno predstavljamo: gostujoči operacijski sistem, ki teče v virtualnem stroju, se ne zaveda, da so njegovi viri prevzaprav navidezni (virtualni). Hipervizor poskrbi, da se ukazi, ki jih gostujoči operacijski sistem pošilja na navidezno strojno opremo, pravilno izvedejo na fizični strojni opremi. Pri tem mora biti v prvi vrsti poskrbljeno za popolno izoliranost hipervizorja od virtualnega stroja, ki ga poganjamo.

Idealno je popolna virtualizacija podprta s strani **strojne opreme**, ki omogoča omenjeno izolacijo s pomočjo posebnih virtualnih kontekstov. Znotraj virtualnega konteksta sme gostujoči operacijski sistem izvrševati poljubne privilegirane ukaze, za katere strojna oprema (mikroprocesor) sama poskrbi, da ne posegajo v delovanje hipervizorja. Privilegirani procesorski ukazi (pravimo jim tudi sistemski ukazi, na primer upravljanje vhodno-izhodnih naprav, branje iz zaščitenega dela pomnilnika), so namreč tisti, ki so potencialno nevarni za hipervizorja (ang. leakage of priviledged state [2]).



Slika 3.3: Popolna virtualizacija (*levo*) omogoča, da gostujoči operacijski sistem v virtualnem stroju teče popolnoma enako, kot bi tekel na resničnem stroju. To dosežemo bodisi z neposredno podporo strojne opreme, bodisi s tehniko ponovnega prevajanja. Paravirtualizacija (*desno*) zahteva modifikacijo gostujočega operacijskega sistema s podporo za uporabo paravirtualnega vmesnika, preko katerega ji hipervizor omogoča poganjanje privilegiranih procesorskih ukazov.

Popolna virtualizacija je možna tudi brez podpore v strojni opremi, vendar mora v tem primeru hipervizor sam poskrbeti za ustrezno zaščito ob izvajanju privilegiranih ukazov. Za to se uporabi pristop imenovan **dinamično ponovno prevajanje binarne kode** (ang. dynamic recompilation, DRC, včasih tudi binary translation, BR), ki strojno kodo ukazov gostujočega operacijskega sistema v realnem času prevaja v novo strojno kodo z emuliranimi privilegiranimi ukazi [2]. Pozorni bralec se na tem mestu vpraša, če hipervizor, ki ukaze virtualnega stroja izvršuje po principu prestrezi-dekodiraj-izvedi, morda ne krši lastnosti *učinkovitosti*, ki jo mora hipervizor izpolnjevati po definiciji. Odgovor je da in ne: hipervizor, ki uporablja metodo DRC sicer ustvarja dodatni sloj med procesorjem in gostujočim operacijskim sistemom, vendar je metoda DRC tako učinkovita, da ta lastnost vseeno velja [2].

Za popolno virtualizacijo je možna strojna podpora tudi neposredno v vhodno-izhodnih napravah (ang. direct I/O), ne le v procesorju. Specifikacija PCIe, preko katere procesor komunicira z vhodno-izhodnimi napravami, specificira tudi razširitev SR-IOV (ang. single-root I/O virtualization)[8], ki ločuje fizične funkcije naprave od podatkovnih. SR-IOV s tem poskrbi, da lahko isto fizično vhodno-izhodno napravo sočasno uporablja desetine izoliranih virtualnih strojev, pri čemer so podatki popolnoma ločeni [10].

Po drugi strani pa je vhodno-izhodne naprave brez strojne podpore za virtualizacijo potrebno emulirati (ang. Full Device Emulation). To pomeni, da mora hipervizor programsko posnemati delovanje naprave tako, da gostujoči operacijski sistem misli, da on edini uporablja izbrano napravo. Hipervizor s pomočjo programskih pasti zazna dostope do naprave in nanje odgovarja z emuliranimi podatki. Glavni problem emulacije so zmogljivostne izgube, saj proženje in lovljenje programskih pasti po nepotrebnem dodaja procesorske cikle.

Hipervizor se pri popolni virtualizaciji osredotoča na čim prepričljivejše posnemanje fizičnih naprav. S tem omogoči poganjanje nedotaknjenih gostujočih operacijskih sistemov, ki so bili razviti za poganjanje na fizičnih napravah. Vendar popolna virtualizacija z vse bolj razširjenimi računalniškimi

oblaki izgublja smisel. Vse več operacijskih sistemov je namreč razvitih prav za delovanje na virtualnih virih, pri čemer se z uporabo paravirtualizacije izognejo potrebi po dragem posnemanju fizičnih naprav.

Paravirtualizacija

Popek et al. v [24] od hipervizorja zahtevajo lastnost *ekvivalence izvajanja*, kar pomeni, da mora biti izvajanje gostujočega operacijskega sistema znotraj virtualnega stroja ekvivalentno, kot če bi ga izvajali neposredno na strojni opremi. Tej zahtevi ustrežejo hipervizorji, ki podpirajo popolno virtualizacijo. Zanje je značilna odvisnost od podpore v strojni opremi na eni strani oz. potreba po dinamičnem ponovnem prevajanju binarne kode na drugi strani - vse samo zaradi zagotavljanja pravilnega izvajanja privilegiranih sistemskih ukazov.

Paravirtualizacija onemogoči izvajanje privilegiranih strojnih ukazov in namesto njih gostujočemu operacijskemu sistemu omogoči alternativni način za doseganje ekvivalentnega učinka z uporabo **paravirtualnega vmesnika**. Hipervizor preko paravirtualnega vmesnika od gostujočega operacijskega sistema prejme zahtevo po izvršitvi sistemskega klica, imenovano hiperklic (ang. hypercall). Dejanski sistemski klic, na primer dodelitev pomnilnika, nato izvede hipervizor in gostujočemu operacijskemu posreduje rezultat.

Poudarimo dve pomembni lastnosti paravirtualizacije. Prvič, gostujoči operacijski sistem mora znati uporabljati paravirtualni vmesnik namesto privilegiranih procesorskih ukazov, torej mora biti posebej prilagojen za izvajanje v virtualnem okolju. Povedano drugače, gostujoči operacijski sistem se mora zavedati, da teče v virtualnem okolju, in sodelovati s hipervizorjem pri čim učinkovitejši virtualizaciji. Druga pomembna lastnost je, da se sistemski ukazi izvajajo na hipervizorju. Gostujoči operacijski sistem naenkrat ne uporablja več svojega virtualnega procesorja za sistemske ukaze, ampak jih zanj izvršuje hipervizor. Upravljanje sistemskih virov preko hipervizorja prinaša performančne prednosti, saj je ravno hipervizor tisti del sistema, ki ima nad viri najboljši pregled.

Emulacija vhodno-izhodnih naprav je pri paravirtualizaciji drugačna kot pri popolni virtualizaciji. Pri slednji hipervizor programsko čim bolj prepričljivo[1] posnema delovanje fizične naprave in s tem omogoči, da gostujoči operacijski sistem uporabi nespremenjene gonilnike zanjo. Rezultat je prepričljiva, vendar neučinkovita navidezna naprava, ki jo je možno uporabljati z obstoječimi gonilniki. Paravirtualizacija pa definira nov model naprave, ki je prilagojen za emulacijo. Hipervizor ne posnema več obstoječe naprave, temveč čimbolj učinkovit približek zanjo. Rezultat je neprepričljiva, vendar izjemno učinkovita navidezna naprava, za uporabo katere je potreben poseben gonilnik. Model takšnega gonilnika je na primer **virtio**, ki ga podrobneje predstavimo v nadaljevanju.

Virtio

Virtio [25] je model za poenoteno realizacijo paravirtualnih vhodno-izhodnih naprav. Definira način zaznavanja virtualnih naprav, njihove konfiguracije, in protokol prenosa podatkov med virtualno napravo in gostujočim operacijskim sistemom. Prenos podatkov je realiziran s pomočjo posebne vrste (ang. virtqueue), na katero sta povezana hipervizor, ki emulira napravo, in gostujoči operacijski sistem, ki napravo uporablja.

Model *virtio* je sestavljen iz dveh osnovnih delov: gonilnikov na gostujočem operacijskem sistemu (ang. front-end drivers) in gonilnikov v hipervizorju (ang. back-end drivers). Prvi omogočajo gostujočemu operacijskemu sistemu, da se poveže na t. i. *virtqueue* in začne komunikacijo z napravo. Gonilniki v hipervizorju pa poenotijo vmesnik emulirane naprave.

Virtio-net [25] je konkreten primer uporabe modela *virtio* za emulacijo omrežne kartice. Uporablja eno vrsto *virtqueue* za prenos podatkov od naprave do gostujočega operacijskega sistema in še eno vrsto *virtqueue* za prenos v obratno smer.

[1] V tem kontekstu *prepričljivo* pomeni, da uporabnik naprave ne opazi razlike v uporabi.

3.2 Virtualizacija z vsebniki

Tehnologija vsebnikov je pristop k podpori virtualizacije brez uporabe hipervizorja [28, 30]. Virtualizacijske entitete, vsebniki, si delijo jedro operacijskega sistema z gostiteljem in so med seboj izolirani s pomočjo podpore operacijskega sistema za nadzorne skupine (ang. control groups) in imenske prostore jedra (ang. kernel namespaces).

Vsebnik ustvarimo tako, da na gostiteljevem sistemu ustvarimo nov proces. Z ukazom **chroot** določimo korenski direktorij datotečnega sistema tega procesa in ga s tem izoliramo od ostalih. Nato z ukazom **cgroups** ustvarimo nadzorno skupino, s katero nastavimo omejitev porabe sistemskih virov (procesorja, pomnilnika in drugih) za ta proces. Sledi ukaz **unshare** za nastavitev imenskih prostorov jedra, ki izolirajo njegove sistemske vire od drugih procesov. Primeri imenskih prostorov so *pid namespace*, ki procesu dodeli enolični identifikator, *net namespace*, ki procesu dodeli lastne omrežne vire (npr. *iptables*, *loopback interface*), *IPC namespace*, ki procesu dodeli lastne semaforje in sporočilne vrste, *mnt namespace*, ki procesu dodeli lastno točko za priklapljanje, ter *UTS namespace*, ki dodeli procesu lastno konfiguracijo *hostname*. Zadnji korak konfiguriranja procesa je izvršitev ukaza za zagon uporabniške aplikacije (ta ukaz je definiran s strani uporabnika, na primer `python manage.py runserver` za zagon spletnega strežnika), prav tako znotraj omenjenega procesa. Tako konfiguriran proces imenujeno vsebnik.

Vsebnik je torej proces, ki izvršuje ukaz za poganjanje aplikacije in teče znotraj izoliranega imenskega prostora. Aplikacija lahko ustvari poljubno število podprocesov, ki vsi ostanejo znotraj istega imenskega prostora. Tudi ti podprocesi so zato del vsebnika in zanje veljajo vse zgoraj naštete omejitve.

Poglavitna prednost virtualizacije z vsebniki v primerjavi z virtualizacijo s hipervizorjem je ta, da vsebnik teče znotraj gostiteljevega operacijskega sistema in uporablja njegove izvorne ukaze [3]. Tabela 3.1, povzeta po [9], prikazuje primerjavo lastnosti virtualnih strojev z vsebniki. Vsebniki uporabljajo skupno jedro operacijskega sistema, ki je že za potrebe gostitelja ves čas naloženo v fizičnem pomnilniku. To po eni strani pomeni, da se prihrani

na prostoru, po drugi strani pa omogoča bistveno krajši zagonski čas, kot če bi morali ob zagonu najprej naložiti jedro z diska v pomnilnik. Z uporabo izvornih ukazov gostitelja se izognemo tudi potrebi po ponovnem prevajanju binarne kode (glej razdelek 3.1.2), saj se privilegirani ukazi lahko nemoteno izvajajo.

Medtem ko virtualni stroji med seboj komunicirajo izključno preko omrežja, se vsebniki lahko pogovarjajo tudi preko signalov in vtičnikov. Virtualni stroji so namreč popolnoma izolirani drug od drugega, pri vsebnikih pa je možna souporaba poljubnih podmap in posledično komunikacija preko njih. Slednje odpira vprašanje varnosti. Varnost virtualnih strojev zavisi od hipervizorja, ki je običajno zelo zanesljiv. Varnost vsebnikov pa zavisi od uporabe imenskih prostorov in nadzornih skupin, ki so nezanesljive, saj njihova varnost zavisi od vsakokratne pravilne konfiguracije procesa. Varnost je tako največja težava, s katero se sooča virtualizacija z vsebniki.

3.3 Tehnologija unikernel

Unikernel je virtualizacijska enota, ki ima lastnosti tako virtualnega stroja, kot tudi vsebnika. Na virtualni stroj nas spominja, ker za svoje delovanje potrebuje hipervizorja in deluje neodvisno od operacijskega sistema gostitelja. Obenem pa je majhne velikosti (nekaj MB) in ima kratek zagonski čas, kar je sicer tipična lastnost vsebnikov.

Pavlicek Russel je v [22] uporabil naslednjo definicijo unikernela:

Definicija 3. *UNIKERNEL je specializiran sistemski posnetek, ki podpira en sam pomnilniški naslovni prostor in je zgrajen z uporabo operacijskega sistema v vlogi knjižnice.*

Z drugimi besedami, unikernel je majhen virtualni stroj brez operacijskega sistema, v njem se izvaja izključno aplikacija. Zgradimo ga tako, da minimalni nabor funkcionalnosti operacijskega sistema zapečemo kar v aplikacijo samo. Edini proces, ki teče znotraj unikernela, je aplikacija. Slednja

Tabela 3.1: Primerjava lastnosti virtualnih strojev z vsebniki (vir: [9]).

PARAMETER	VIRTUALNI STROJ	VSEBNIK
GOSTUJOČI OPERACIJSKI SISTEM	vsak stroj ima svoj virtualni pomnilnik, v katerega ima naložen OS	vsebniki si delijo jedro OS, ki je naloženo v fizičnem pomnilniku
PORABA PROSTORA	potrebuje veliko prostora, saj vsebuje kopijo operacijskega sistema	potrebuje malo prostora, saj ne vsebuje jedra operacijskega sistema
ZAGONSKI ČAS	nekaj minut	nekaj sekund
UČINKOVITOST	dodatni režijski čas za ponovno prevajanje binarne kode	enaka učinkovitost, kot če bi izvajali na gostitelju
KOMUNIKACIJA	preko virtualne mrežne kartice	preko signalov in vtičnikov
IZOLIRANOST	stroji so popolnoma izolirani drug od drugega	vsebniki si lahko delijo poljubne podmape
VARNOST	odvisno od hipervizorja	uporaba imenskih prostorov in nadzornih skupin

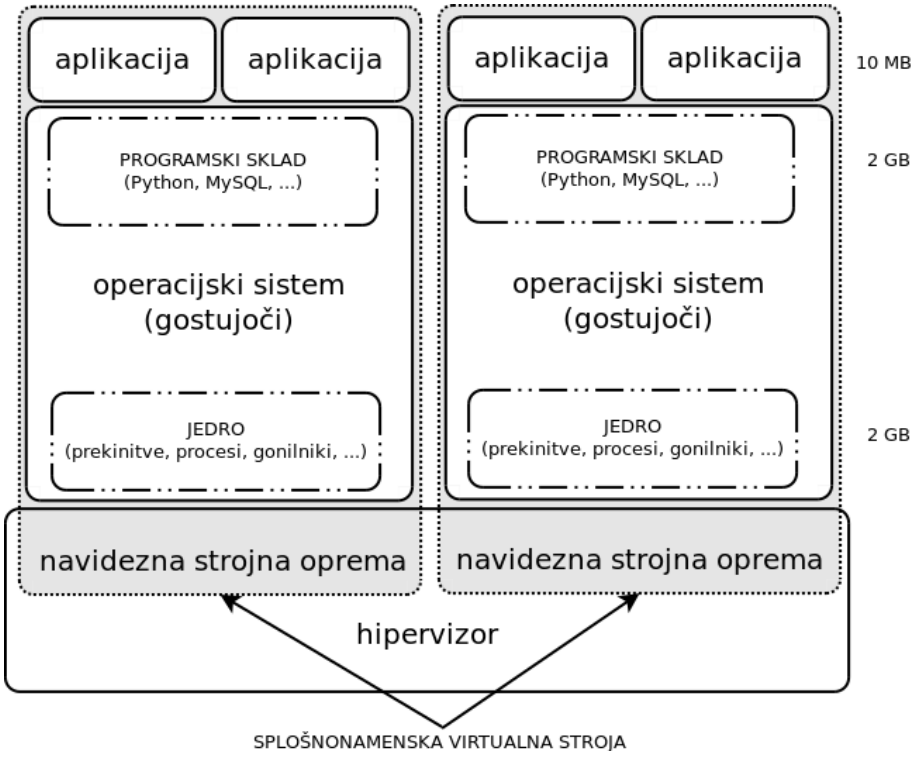
po potrebi uporabi knjižnico, ki vsebuje funkcije operacijskega sistema (ang. library operating system).

3.3.1 Motivacija za razvoj tehnologije unikernel

Razvijalci oblačne infrastrukture običajno stremijo k izpopolnjevanju tehnologije za virtualizacijo. Čim učinkovitejša je uporaba strojnih virov, tem zmogljivejši je oblak. To nas je pripeljalo k skoraj popolnim hipervizorjem, ki znajo učinkovito izkoristiti dano strojno opremo. Tudi programska oprema za upravljanje računalniškega oblaka je zrela in zna transparentno uporabljati računsko moč množice fizičnih strojev, ne le posameznega. Zdi se, da je računalniški oblak dosegel končno stopnjo razvoja.

Oblačne infrastrukture so torej sposobne odlično poganjati splošnonamenske virtualne stroje, v katerih nato poganjamo aplikacije (glej sliko 3.4). Pojem *splošnonamenski virtualni stroj* označuje virtualni stroj, v katerem teče gostujoči operacijski sistem z nameščenim programskim skladom (ang. software stack) in je tipično velikosti nekaj GB. Namen splošnonamenskega virtualnega stroja je poganjanje ene ali več uporabniških aplikacij, ki so tipično zelo majhne, le nekaj MB. Opazimo, da pridemo do paradoksne situacije: uporabnik oblačne infrastrukture virtualizira celoten splošnonamenski virtualni stroj, velik nekaj GB, da potem na njem požene svojo aplikacijo, ki uporablja le skromno podmnožico ponujenih funkcionalnosti. Večino dodeljenih sistemskih virov tako zasedemo po nepotrebnem.

Uporaba splošnonamenskih virtualnih strojev ni optimalna niti pri zagotavljanju **varnosti**. Če je na virtualnem stroju nameščenih na stotine programov, je nemogoče zagotoviti, da nobeden od njih nima varnostne luknje. Tudi če posodobimo vse programe z najnovejšimi varnostnimi posodobitvami in aktiviramo varnostne mehanizme (npr. SELINUX), še vedno ne moremo dovolj dobro testirati sistema, da bi preverili, ali smo zaščiteni vsaj pred najbolj znanimi napadi na sistem. Prav tako ne moremo izvesti formalne analize tveganja, saj je splošnonamenski sistem preveč kompleksen, da bi lahko izvedli vsa potrebna testiranja. Potem so tu še bolj sofisticirani napadi, kot je na



Slika 3.4: Splošnonamenski virtualni stroj za svoje delovanje zahteva veliko sistemskih virov, predvsem diska. Običajno veliko več, kot aplikacija, ki jo na njem poganjamo.

primer napad preko gonilnikov. Novembra 2015 je podjetje RedHat razkrilo napad preko gonilnika za branje gibkih diskov (t. i. 'disket'), kjer je lahko napadalec izvedel poljuben privilegiran ukaz na virtualnem stroju, če je le imel dostop do pravih vhodno-izhodnih vrat[2]. Čeprav se gibkih diskov že nekaj let ne uporablja več, so njihovi gonilniki še vedno del splošnonamenskih operacijskih sistemov in po nepotrebnem predstavljajo varnostno grožnjo.

Uporaba splošnonamenskih virtualnih strojev torej prinaša dve veliki težavi: nepotrebno trošenje sistemskih virov in veliko površino napada. Zato se je pojavila potreba po novi virtualizacijski enoti, ki bi uporabljala le toliko sistemskih virov, kot jih je potrebnih za delovanje aplikacije, in bi pri tem čim bolj zmanjšala tveganje za nastanek varnostnih incidentov.

Tehnologija vsebnikov, ki smo jo opisali v poglavju 3.2, dobro naslovi potrebo po optimizaciji porabe sistemskih virov. Vsebniki si delijo jedro operacijskega sistema z gostiteljem in omogočajo souporabo programskih skladov. Ko uporabnik zažene svojo aplikacijo, se tako po nepotrebnem ne zasede velika količina sistemskih virov za njeno režijo, temveč le toliko, kolikor je za delovanje aplikacije potrebno. Ko poženemo deset vsebnikov, se zasede toliko sistemskih virov, kot jih je potrebno za poganjanje desetih aplikacij. To je zelo učinkovito v primerjavi s splošnonamenskimi virtualnimi stroji, ki poleg aplikacijskih procesov poganjajo še na stotine sistemskih procesov. Ko poženemo deset splošnonamenskih virtualnih strojev za poganjanje aplikacije, se zato zasede toliko sistemskih virov, kolikor jih je potrebnih za poganjanje desetih operacijskih sistemov in desetih aplikacij.

Kljub temu pa vsebniki ne naslovijo potrebe po izboljšanju varnosti. Varnostna grožnja za vsebnike je natanko enako velika kot pri splošnonamenskih virtualnih strojih, saj tečejo v operacijskem sistemu gostitelja, ki je splošnonamenski. Lahko bi celo rekli, da so vsebniki še bolj izpostavljeni varnostnim luknjam, ker njihova varnost v celoti zavisi od tega, kako jih administrator konfigurira ob zagonu. V razdelku 3.2 smo namreč opisali, da vsebnik ni nič drugega kot posebej konfiguriran proces. Če kaj pozabimo pri konfigu-

[2]https://access.redhat.com/blogs/product-security/posts/1976633

raciji, je vsebnik popolnoma brez zaščite. Kot zanimivost omenimo, da se razvijalci vsebnikov te težave zavedajo, vendar zaenkrat kaže, da jo je možno zaobiti le na račun enostavnosti uporabe. Povedano drugače, če v vsebnike vnesemo napredne varnostne mehanizme, jih zaradi kompleksnosti verjetno ne bo nihče več uporabljal. Vsebniki zato le deloma zadostijo virtualizacijski enoti, ki bi učinkoviteje in varneje izvajala uporabniško aplikacijo.

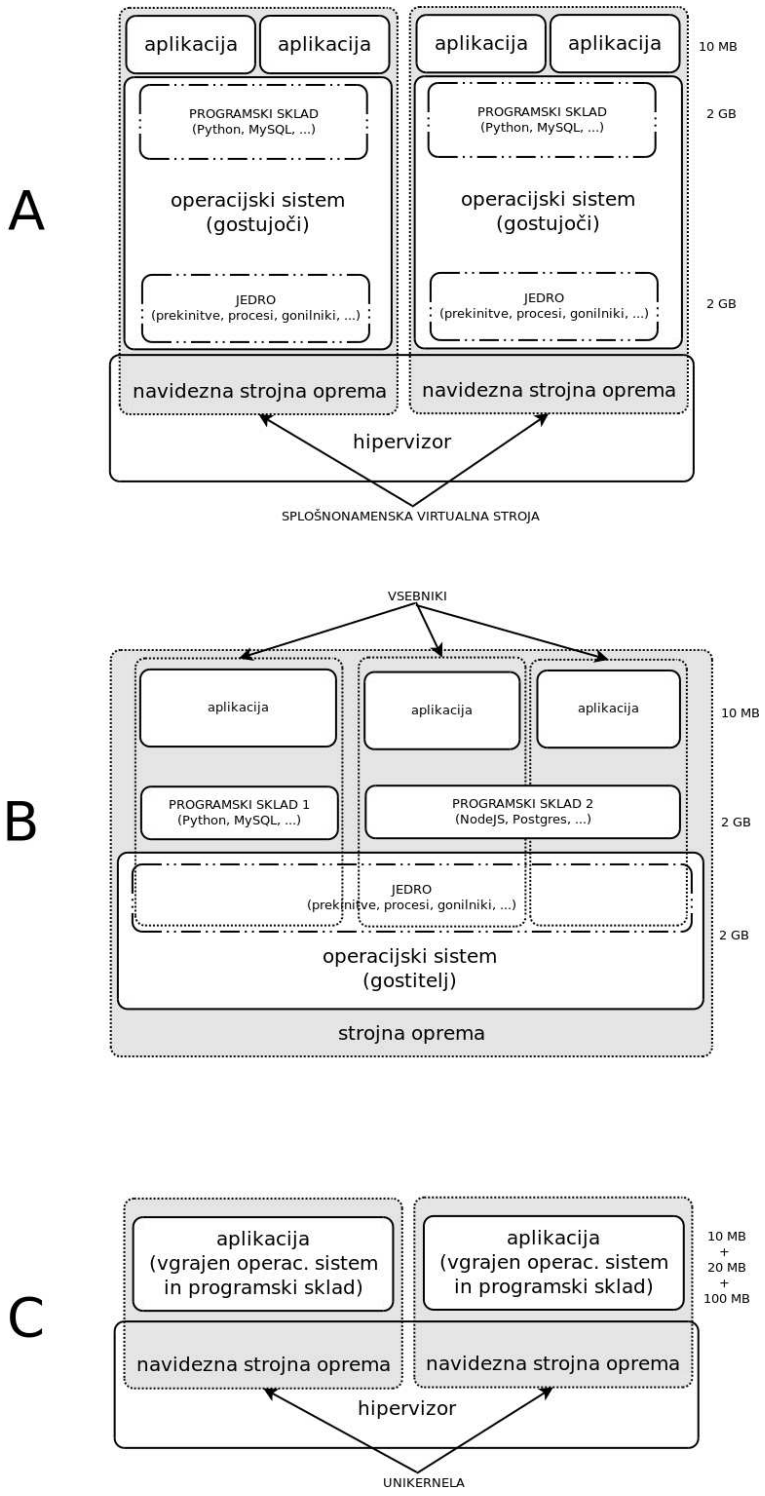
Pravi odgovor pa je tehnologija unikernel, ki predstavi radikalno spremenjen pogled na operacijski sistem in programski sklad. Slika 3.5 prikazuje arhitekturno shemo splošnonamenskega virtualnega stroja, vsebnika in unikernela. Unikernel je izjemno majhna virtualizacijska enota, v nekaterih implementacijah celo neverjetnih 200 kilobajtov. Vsebuje namreč izključno funkcionalnosti, ki so potrebne za poganjanje aplikacije, in ničesar drugega, niti operacijskega sistema. Tistih nekaj funkcionalnosti, ki bi jih aplikacija potrebovala od operacijskega sistema, so kar sestavni del aplikacije. Podobno velja za programski sklad. V aplikacijo so vključeni le tisti programi in knjižnice, ki jih aplikacija potrebuje.

Z odvzemanjem nepotrebnih funkcionalnosti iz operacijskega sistema in programskega sklada bistveno zmanjšamo možnosti napada. Ne le, da je manj možnosti, kako bi se napadalec lahko povezal na unikernel, tudi če se mu uspe povezati, nima tam na voljo nobenih orodij, s katerimi bi lahko karkoli počel. Unikerneli tako že s svojo arhitekturno zasnovo zagotovijo bistveno višji nivo varnosti kot splošnonamenski virtualni stroji ali vsebniki.

Unikernel se torej izkaže kot primerna virtualizacijska enota, ki reši tako paradoks velikosti virtualnega stroja v primerjavi z aplikacijo, kot tudi zahtevo po večji varnosti. Radovedni bralec se na tem mestu morda že sprašuje, kako pravzaprav sploh deluje unikernel. Na vprašanje odgovorimo v naslednjem poglavju.

3.3.2 Delovanje unikernela

V unikernelu vedno teče le *ena* aplikacija, ki hkrati prevzame vlogo operacijskega sistema. Iz tega sledi, da ni več potrebe po deljenju pomnilniškega



Slika 3.5: Arhitektura splošnonamenskega virtualnega stroja (A), vsebnika (B) in unikernela (C).

prostora na uporabniški del in jedrni del (glej sliko 3.6). Del pomnilnika, ki ga uporablja operacijski sistem, namreč v običajnih sistemih pred uporabnikom posebej ščitimo. Uporablja se za dostop do datotečnega sistema, dostop do vhodno-izhodnih naprav in omrežja ter za upravljanje s procesi. Z omejitvijo dostopa preprečimo, da bi ena sama neustrezno sprogramirana aplikacija vplivala na delovanje preostalih aplikacij. Pri unikernelih s tem nimamo težav, saj v njih vedno teče natanko ena aplikacija. Tudi če je ta aplikacija nevarna, ne more škodovati drugim aplikacijam, ker jih preprosto ni. Zato unikerneli lahko uporabljajo en sam naslovni prostor. Opazimo tudi, da ker v unikernelu teče le en proces, aplikacija, ni več potrebe po logiki za upravljanje s procesi. Prav tako v unikernelu ni upravljanja z uporabniki. Aplikacija, ki teče v unikernelu, je samozadostna. Vsebuje tako:

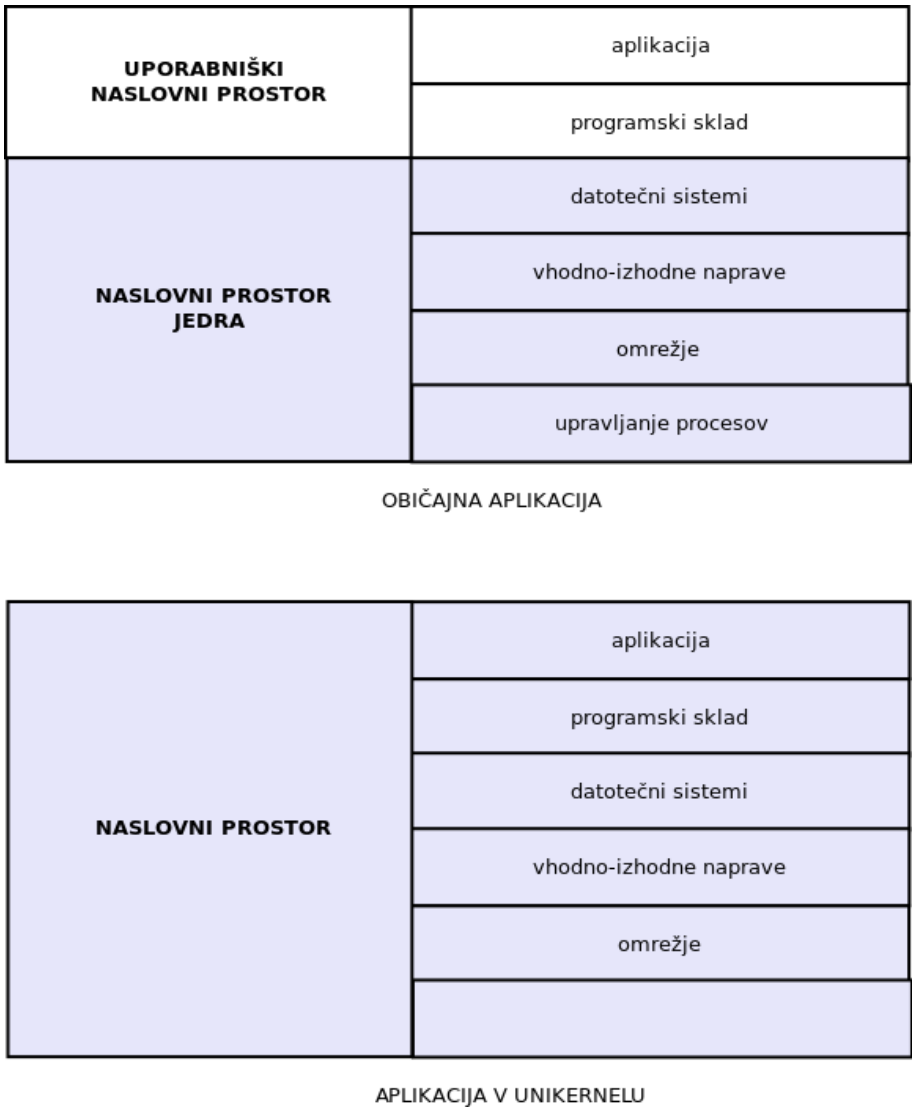
1. visokonivojsko aplikacijsko logiko (na primer kodo za spletni strežnik, napisano v jeziku JavaScript),
2. celoten programski sklad (na primer izvajalno okolje NodeJS v obliki deljenega objekta), in tudi
3. nizkonivojske funkcije (na primer za odpiranje datotek).

Za poganjanje tako potrebujemo le še datotečni sistem, kamor shranimo datoteko z aplikacijo, ter (navidezno) strojno opremo, ki požene aplikacijo.

3.3.3 Omejitve unikernela

Na splošnonamenskih virtualnih strojih lahko poganjamo poljubno kompleksno aplikacijo. Unikerneli pa so specializirani in temeljijo na določenih predpostavkah, ki jih vse aplikacije ne izpolnjujejo. Oglejmo si te predpostavke.

V unikernelu lahko teče **samo en proces**. Ukaz `fork()`, s katerim v običajnih sistemih ustvarjamo podprocese, je prepovedan in povzroči ustavitev unikernela. Unikernel namreč že po definiciji ne more poganjati več procesov, saj nima podpore za delo z njimi. Na to omejitev moramo v praksi najbolj paziti. Če naša aplikacijska logika ali katerakoli knjižnica, ki jo logika



Slika 3.6: Delitev naslovnega prostora pri unikernelih ni več potrebna. Prav tako ni potrebe po upravljanju s procesi, saj v unikernelu vedno teče en sam proces (aplikacija).

uporabi, kliče ukaz `fork()`, bo unikernel prenehal delovati. Kot zanimivost omenimo, da unikerneli kljub omejitvi na en proces vsebujejo polno podporo za delo s poljubnim številom programskih niti.

V unikernelu obstaja **samo en uporabnik**, `root`. Če bi unikerneli nudili podporo za več uporabnikov, bi se njihova kompleksnost (in s tem velikost), močno povečala. V preprostem sistemu, kjer je dovoljen samo en proces, se podpora za več uporabnikov niti ne zdi smiselna. Te omejitve pa se moramo zavedati, kadar želimo svojo aplikacijo pretvoriti v unikernel. Če aplikacija zahteva podporo za več uporabnikov, nam je ne bo uspelo poganti.

Unikernel **ni namenjen razhroščevanju** aplikacije, temveč njenemu poganjanju. Predpostavlja se, da aplikacijo najprej razvijemo v splošnonamenskem operacijskem sistemu in jo v unikernelu le poženemo. Pisanje dnevniških zapiskov seveda deluje tudi v unikernelu in nam služi kot edini način za odkrivanje napak, zato je zaželjeno, da naša aplikacijska logika to možnost izčrpno uporablja.

Lahko se zgodi, da naša aplikacija potrebuje funkcijo operacijskega sistema, ki za izbrano implementacijo unikernela **trenutno še ni podprta**. V tem primeru se moramo obrniti na avtorja unikernela in ga prositi, če jo lahko implementira za nas.

3.3.4 Implementacije unikernelov

Unikernel ustvarimo tako, da osnovni aplikacijski kodi dodamo podmnožico funkcionalnosti operacijskega sistema. Obstaja več načinov, kako to naredimo, torej katero podmnožico funkcionalnosti izberemo. V nadaljevanju predstavimo nekaj implementacij unikernelov. Med seboj se ločijo predvsem po izbiri podmnožice funkcionalnosti operacijskega sistema in po implementaciji posamezne sistemske funkcije. Nekateri unikerneli izhajajo iz splošnonamenskih operacijskih sistemov in so le njihove okleščene različice, pri drugih pa so sistemske funkcionalnosti implementirane povsem na novo, tako da lahko kar najbolje izkoristijo opisane lastnosti unikernelov.

MirageOS [17] je ena prvih implementacij unikernela, začetna verzija je bila objavljena konec leta 2013. Razvili so jo na angleški univerzi Cambridge in podpira poganjanje aplikacij napisanih izključno v programskem jeziku OCaml. Unikernel MirageOS je specializiran za poganjanje na hipervizorju Xen, saj temelji na uporabi njegove implementacije paravirtualizacije. Ker poleg paravirtualizacijskega gonilnika za Xen ne potrebuje nobenih drugih gonilnikov, je zelo majhen - znan primer je unikernel s popolnoma delujočim strežnikom DNS, velik zgolj 184 KB [22].

Unikernel tipa MirageOS ustvarimo s pomočjo množice pomožnih programov in skript, ki jih moramo pravilno konfigurirati. V postopku izgradnje unikernela moramo prevajalniku namreč povedati, katero podmnožico funkcionalnosti operacijskega sistema želimo. Zgrajeni unikernel tako ne vsebuje nobenih dodatnih funkcionalnosti, zato je zelo majhen.

Problem unikernela MirageOS je njegova omejenost na jezik OCaml in na hipervizor Xen. Če naša aplikacija ni napisana v tem jeziku, jo moramo ponovno implementirati, kar je nepraktično. Neprijetna lastnost je tudi kompleksnost ustvarjanja unikernela. Uporabnik si mora najprej namestiti ustrezne programe za prevajanje, nato mora za njih nastaviti ustrezno konfiguracijo, kar je zamudno in polno frustracij.

IncludeOS [6] je unikernel, ki ga razvijajo na norveški univerzi *Oslo and Akershus University* in trenutno še nima uradne različice. V njem lahko poganjamo aplikacije napisane v programskem jeziku C++. Podobno kot MirageOS tudi IncludeOS predpostavlja, da ga bo poganjal hipervizor s podporo paravirtualizaciji. Vendar so se avtorji tu odločili za implementacijo gonilnika po splošnem modelu *virtio*. V razdelku 3.1.2 smo omenili, da je *virtio* splošen model naprave pri paravirtualizaciji in ni vezan na specifičen hipervizor. To pomeni, da lahko IncludeOS poganjamo na poljubnem hipervizorju s podporo paravirtualizaciji po modelu *virtio*, na primer KVM, Virtualbox ali QEMU.

Unikernel IncludeOS ustvarimo s pomočjo pomožnega programa. Konfiguracija je enostavnejša kot pri MirageOS, saj zna ta pomožni program

(gre za nadgradnjo orodja GCC) na podlagi aplikacijske kode sam izbrati potrebno podmnožico sistemskih funkcionalnosti. Unikernel IncludeOS je podobno majhen kot MirageOS - popolnoma delujoč strežnik DNS je velik le 158 KB [6].

Rumprun [11] je unikernel, ki potrebno podmnožico funkcionalnosti operacijskega sistema implementira s pomočjo orodja za testiranje gonilnikov naprav imenovanega *Rump kernel*. Orodje v splošnem omogoča pretvorbo poljubnih gonilnikov v statično prevedeno modularno različico. Slednje omogoča aplikaciji, ki teče znotraj splošnonamenskega operacijskega sistema, da uporablja izbrane gonilnike ne glede na gonilnike, ki jih sicer uporablja operacijski sistem. Zanimivo je predvsem to, da gonilniki pri pretvorbi z orodjem ostanejo nespremenjeni, kar omogoča popolnoma enako delovanje, kot če bi jih uporabljali neposredno.

Unikernel Rumprun vsebuje gonilnike, ki so statično prevedeni z orodjem Rump kernel, poleg tega pa še preprosto povezovalno logiko, ki nadomesti manjkajoče funkcije operacijskega sistema. Pomembno je, da ta povezovalna logika sledi standardu POSIX (ang. Portable Operating System Interface), torej uporablja isti vmesnik kot splošnonamenski operacijski sistemi. Aplikacije, ki jo želimo poganjati v unikernelu, zato ni potrebno prilagajati. Isto kodo lahko poženemo na splošnonamenskem operacijskem sistemu ali v Rumprun unikernelu, če le ustreza omejitvam, ki smo jih navedli v razdelku 3.3.3. Tako je možno pognati aplikacije napisane v programskih jezikih C, C++, Erlang, Go, Java, JavaScript, Python, Ruby, Rust.

Unikernel Rumprun ustvarimo s pomočjo pomožnih programov in skript. Konfiguracija je kompleksna, saj moramo izbrati nabor gonilnikov, ki jih želimo uporabiti. Rezultat je unikernel, velik okrog 4 MB več kot aplikacija sama. To je sicer več kot pri unikernelih MirageOS in IncludeOS, vendar še vedno bistveno manj, kot če bi uporabili splošnonamenski virtualni stroj. Prednost unikernela Rumprun pred MirageOS in IncludeOS je splošnost uporabe, saj v unikernelu Rumprun lahko poganjamo skoraj poljubno aplikacijo POSIX.

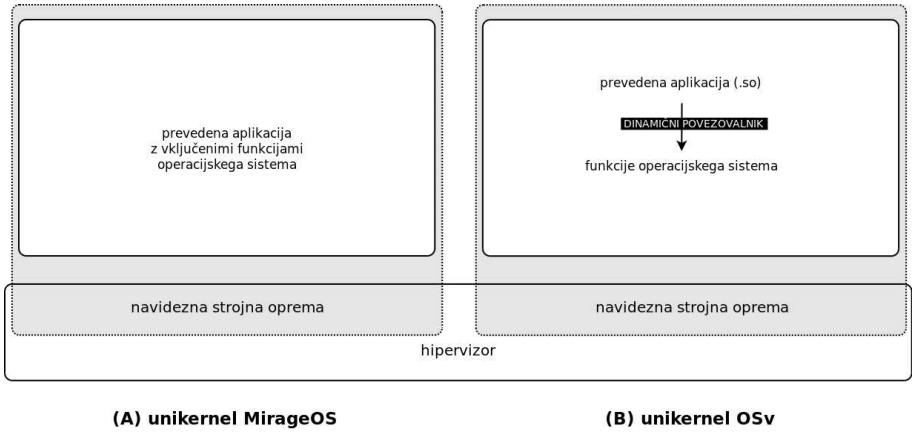
Zanimiva je uporaba znanega programskega sklada LAMP (*Linux*, Apache, MySQL, PHP) s pomočjo treh unikernelov Rumprun. Posamezni unikernel poganja eno komponento sklada, rešitev pa se imenuje RAMP (*Rumprun*, Apache, MySQL, PHP)[3]. RAMP je dokaz splošne uporabnosti unikernela Rumprun, saj z njim lahko poganjamo nespremenjene aplikacije celotnega programskega sklada spletnih strežnikov.

OSv [13] je poseben unikernel, ki se od drugih loči predvsem po tem, da so avtorji potrebno podmnožico funkcionalnosti operacijskega sistema implementirali povsem na novo in pri tem upoštevali tako zakonitosti okolja, v katerem se bo unikernel izvajal (tj. na hipervizorju), kot tudi lastnosti unikernelov (npr. en sam naslovni prostor, en sam proces). Pri ponovni implementaciji so tako uvedli številne optimizacije, zaradi katerih so unikerneli OSv ne le bistveno manjši od splošnonamenskih operacijskih sistemov, temveč tudi hitrejši. Pri tem so uporabili vmesnik POSIX, zato aplikacij ni potrebno posebej prilagajati.

Druga posebnost, po kateri se unikernel OSv bistveno loči od ostalih implementacij unikernela, je uporaba **dinamičnega povezovalnika** (ang. dynamic linker) za poganjanje aplikacije, ki med izvajanjem pravilno poveže aplikacijo z ustreznimi funkcijami na novo implementiranega operacijskega sistema (glej sliko 3.7). Medtem ko je za druge unikernele potrebno aplikacijo prevesti z namenskim prevajalnikom, ki vanjo zapeče funkcije operacijskega sistema, lahko za OSv uporabimo isti prevajalnik kot na splošnonamenskih sistemih. Povedano drugače, če imamo aplikacijo prevedeno že od prej, lahko skopiramo rezultat (datoteke s končnico .so) v unikernel OSv in program bo deloval.

Unikernel OSv zgradimo tako, da vedno uporabimo enako podmnožico (na novo implemeniranih) funkcij operacijskega sistema. Povedano drugače, pri pakiranju naše aplikacije v unikernel ni potrebno izbirati, katere komponente bomo vključili in katerih ne, ker vedno vključimo vse. Tako je konfi-

[3]Namesto spletnega strežnika Apache so avtorji pravzaprav uporabili spletni strežnik Nginx.



Slika 3.7: Unikernel OSv se od drugih loči po tem, da se aplikacija z operacijskim sistemom poveže šele med izvajanjem (s pomočjo dinamičnega povezovalnika). Medtem ko je za MirageOS potrebno aplikacijo prevesti s posebnim prevajalnikom, ki vanjo zapeče funkcije operacijskega sistema, lahko za OSv uporabimo katerikoli prevajalnik oz. že kar prevedeno aplikacijo.

guracija za uporabnika prijaznejša, zato se slednji osredotoča na nastavitve parametrov na nivoju aplikacije, in ne na nivoju operacijskega sistema.

Zakasnitev vključitve funkcij operacijskega sistema k aplikaciji omogoča poenostavitev izgradnje unikernelov, saj lahko funkcije operacijskega sistema prevedemo vnaprej in rezultat shranimo v obliki vnaprej pripravljenega osnovnega unikernela. Ko lokalno prevedemo svojo aplikacijo, jo preprosto dodamo v osnovni unikernel in že je pripravljena za zagon na hipervizorju. Pozorni bralec se bo na tem mestu verjetno vprašal, kako velik je osnovni unikernel, ki vsebuje nabor vseh implementiranih funkcij operacijskega sistema. Odgovor je 8 MB. To je sicer več, kot so veliki drugi tipi unikernelov, vendar še vedno zelo malo v primerjavi s splošnonamenskimi virtualnimi stroji.

3.4 Oblačno ogrodje OpenStack

OpenStack [27] je priljubljena odprtokodna platforma za upravljanje računalniškega oblaka. Njena modularna zasnova omogoča neodvisen razvoj posameznih komponent. Osnovne komponente platforme so OpenStack Nova (za upravljanje vozlišč), OpenStack Neutron (za upravljanje omrežja), OpenStack Glance (za shranjevanje sistemskih posnetkov) in OpenStack Horizon (spletni vmesnik) [23].

OpenStack je specializiran za postavljanje virtualnih strojev in njihovo upravljanje ter medsebojno povezovanje. V osnovi uporablja virtualizacijo s hipervizorjem, obstaja pa tudi delna podpora za virtualizacijo z vsebniki[4].

Osnovni način uporabe ogrodja OpenStack za zagon izbranega sistemskega posnetka je sledeč. Uporabimo brskalnik za dostop do spletnega vmesnika OpenStack Horizon, preko katerega izvedemo vse naslednje korake. S pomočjo ustreznega spletnega obrazca izberemo sistemski posnetek na našem razvojnem računalniku in ga s klikom na gumb prenesemo v zbirko sistemskih posnetkov na ogrodju OpenStack. Iz njega nato ustvarimo instanco virtualnega stroja in jo zaženemo. Vse to storimo s klikom na ustrezen gumb. Zadnji korak je povezava naslova IP, ki je dosegljiv od zunaj, na dobljeno instanco. To storimo z uporabo ustrezne izbire v meniju. Sistemski posnetek je tedaj zagnan in dostopen iz omrežja. Instanco lahko pričnemo uporabljati.

Ogrodje OpenStack seveda podpira še bistveno večji nabor akcij, ki jih lahko izvedemo, in konfiguracij, ki jih lahko nastavimo. Vendar opis vseh presega okvir magistrskega dela, saj smo pri izdelavi uporabljali le osnovni način uporabe. Pri tem smo namesto splošnonamenskih sistemskih posnetkov, ki jih običajno poganjamo na ogrodju OpenStack, uporabili namenske sistemske posnetke, tj. unikernele. Več o tem v nadaljevanju.

[4]https://github.com/openstack/kuryr

Poglavje 4

Testno okolje

Implementirali smo spletno aplikacijo v tehnologiji NodeJS, ki smo jo nato z uporabo tehnologije unikernel pognali na oblačnem ogrodju OpenStack. V tem poglavju predstavimo omenjeno spletno aplikacijo in opišemo naše testno okolje z lokalno postavitvijo oblačnega ogrodja OpenStack. V razdelku 4.3 nato za dano kombinacijo aplikacije in ciljnega okolja izberemo podmnožico primernih implementacij tehnologije unikernel.

4.1 Testna spletna aplikacija Medo

NodeJS [29] je izvajalno okolje namenjeno poganjanju strežniškega dela kode spletnih aplikacij (ang. server-side runtime). Medtem ko je samo okolje NodeJS napisano v kombinaciji programskih jezikov C in C++, mora biti aplikacijska koda spletne aplikacija napisana v programskem jeziku JavaScript. NodeJS se od ostalih spletnih ogrodij loči predvsem po tem, da za sočasno obravnavo množice spletnih zahtevkov ne potrebuje več procesov ali več niti, temveč uporablja povratne klice. Povedano drugače, izvajalno okolje NodeJS že na nivoju ene programske niti poskrbi za sočasnost. Omenjena lastnost je vsekakor dobrodošla v kontekstu unikernelov, ki že po definiciji podpirajo izvajanje zgolj enega procesa. Druga zaželena lastnost okolja NodeJS je ta, da je aplikacijska koda napisana v skriptnem jeziku JavaScript, ki ga ni po-

trebno prevajati. Z vidika unikernelov to pomeni, da če nam uspe pravilno prevesti izvajalno okolje NodeJS, da teče znotraj unikernela, potem bomo brez težav poganjali poljubno aplikacijsko logiko.

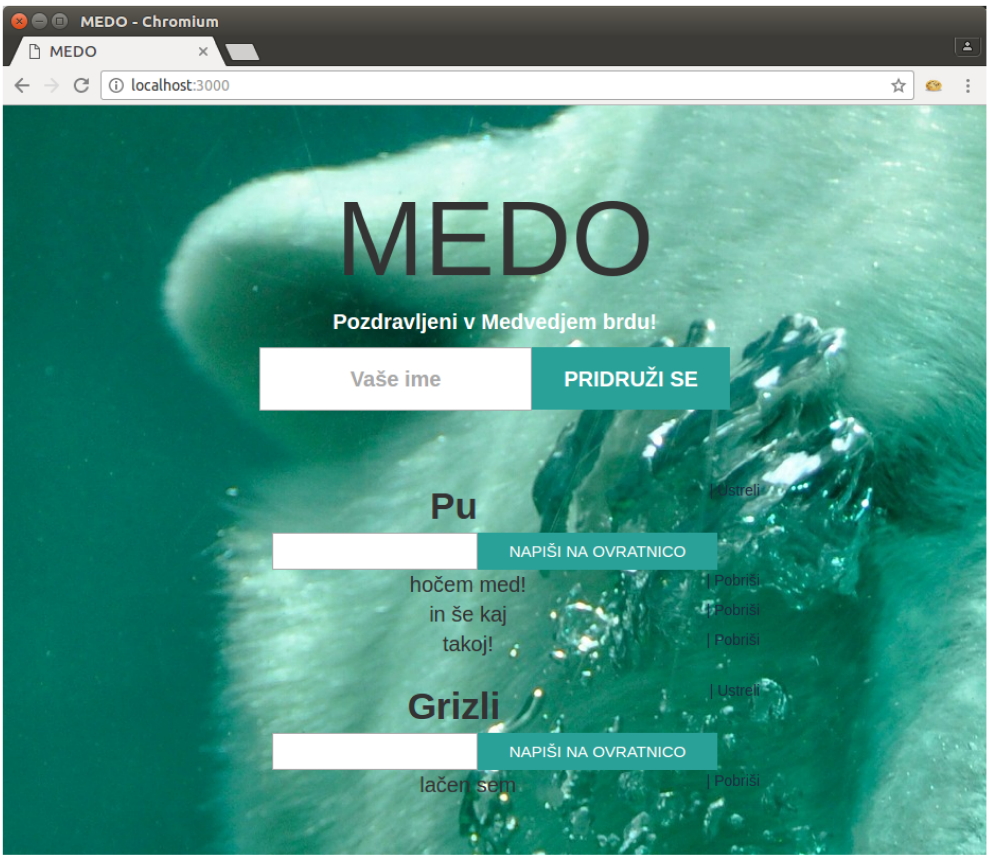
Implementirali smo preprosto, vendar ne trivialno, spletno aplikacijo za okolje NodeJS in jo poimenovali Medo. Aplikacija iz okoljske spremenljivke `PORT` razbere številko vrat na katerih ob zagonu začne streči interaktivno spletno stran s seznamom medvedov. Spletna stran omogoča dodajanje novih objektov na seznam ter brisanje s seznama. Pri tem je pomembno, da se objekti v ozadju shranjujejo v podatkovno bazo MySQL. Kot zanimivost omenimo, da smo za shranjevanje podatkov uporabili knjižnico `sequelize`[1], ki je priljubljena tudi za razvoj kompleksnih aplikacij. Omenimo še uporabo knjižnice `jade`[2], ki nam olajša pisanje predlog spletne strani tako, da nadomesti jezik HTML z uporabniku prijaznejšo sintakso. Z razvojem takšne aplikacije smo se želeli čim bolj približati kontekstu, s kakršnim se na delovnem mestu srečuje razvijalec spletnih aplikacij. Pri ocenjevanju uporabniške izkušnje pri izgradnji unikernela v naslednjih poglavjih se namreč pogosto postavimo v njegovo vlogo.

Slika 4.1 prikazuje posnetek zaslona spletne aplikacije Medo. Aplikacija ponudi vnosno polje za vnos imena novega medveda. Vsakega dodanega medveda lahko bodisi odstranimo, bodisi mu napišemo sporočilo na ovratnico. Vse operacije se izvedejo na strežniški strani in dostopajo do podatkovne baze MySQL. Podatkovna baza je preprosta - vsebuje le eno tabelo za imena medvedov in eno za napise na ovratnicah.

4.2 Postavitev testnega okolja

Testno spletno aplikacijo, opisano v prejšnjem poglavju, želimo z uporabo tehnologije unikernelov pognati na oblačnem ogrodju OpenStack. Ker nismo imeli razpoložljive obstoječe postavitve, smo se odločili za lokalno postavi-

[1] https://github.com/sequelize/sequelize
[2] https://www.npmjs.com/package/jade



Slika 4.1: Posnetek zaslona spletne aplikacije Medo.

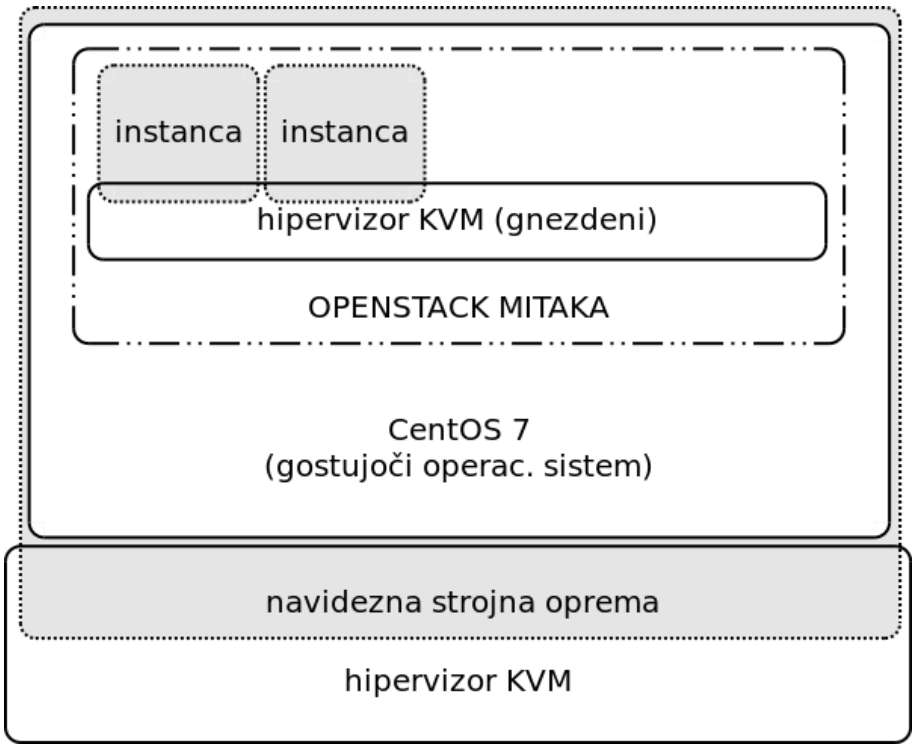
tev celotnega ogrodja OpenStack z uporabo orodja Packstack[3], ki v veliki meri avtomatizira nameščanje komponent ogrodja OpenStack. Z uporabo hipervizorja KVM smo ustvarili splošnonamenski virtualni stroj, v katerega smo namestili operacijski sistem CentOS 7, ki je edini podprt s strani orodja Packstack. Znotraj gostujočega operacijskega sistema CentOS 7 smo z uporabo orodja Packstack namestili najnovejšo uradno podprto verzijo ogrodja OpenStack - OpenStack Mitaka.

Kot zanimivost omenimo, da smo prvotno namesto hipervizorja KVM uporabili hipervizor VirtualBox, vendar se je izkazalo, da ne omogoča gnezdene popolne virtualizacije s podporo v strojni opremi. Instance, ki smo jih pognali na takšni postavitvi ogrodja OpenStack, so morale teči na hipervizorju s programskim izvajanjem popolne virtualizacije, kar je bistveno upočasnilo delovanje. Nato smo ponovili korake namestitve gostujočega operacijskega sistema CentOS 7 (in nato ogrodja OpenStack znotraj njega) v novem virtualnem stroju, ki je tokrat tekel na hipervizorju KVM s kombinacijo uporabe strojno podprte popolne virtualizacije in paravirtualizacije. Tedaj je ogrodje OpenStack lahko uporabilo gnezdeni hipervizor KVM s strojno podprto popolno virtualizacijo, kar je bistveno pospešilo delovanje instanc.

Slika 4.2 prikazuje shemo končne postavitve testnega okolja. Na razvojnem računalniku poganjamo Ubuntu 16.04 LTS, v okrilju katerega teče hipervizor KVM. Gostujoči operacijski sistem CentOS 7 teče na hipervizorju KVM, znotraj njega pa je nameščeno ogrodje OpenStack, ki za poganjanje instanc uporablja gnezdeni hipervizor KVM. Tako globalni kot gnezdeni hipervizor KVM podpirata popolno virtualizacijo s strojno podporo, zato instance, ustvarjene s pomočjo ogrodja OpenStack, tečejo zadovoljivo hitro.

Postavitev ogrodja OpenStack z uporabo orodja Packstack je enostavna z izjemo konfiguracije omrežja. Ker smo želeli, da so z ogrodjem OpenStack ustvarjene instance dosegljive tudi znotraj lokalnega omrežja razvojnega računalnika, smo morali ročno konfigurirati ustrezne mrežne mostove (ang. network bridges). To je kompleksno opravilo in zahteva poglobljeno

[3]https://www.rdoproject.org/install/quickstart



Slika 4.2: Shema postavitve testnega okolja.

poznavanje tako mrežne konfiguracije sistema Ubuntu, kot tudi komponente Neutron ogrodja OpenStack, kar presega obseg magistrskega dela, zato bomo opis slednjega izpustili.

4.3 Izbira implementacije unikernela

Postavimo se zopet v vlogo spletnega razvijalca in poskusimo pognati testno aplikacijo Medo (glej razdelek 4.1) na testni postavitvi ogrodja OpenStack (glej razdelek 4.2) s pomočjo tehnologije unikernel. Prvi korak proti cilju je izbira ustrezne implementacije tehnologije unikernel. Obstaja namreč množica implementacij, ki se med seboj bistveno razlikujejo [22]. Nekateri tipi unikernelov so specializirani za opravljanje točno določene funkcije, drugi za poganjanje aplikacij v določenem programskem jeziku. Obstajajo pa tudi implementacije unikernelov, ki nudijo podporo za poganjanje aplikacije, napisane v poljubnem programskem jeziku. Kateri tip unikernela torej izbrati za poganjanje aplikacije Medo?

Aplikacija Medo za svoje delovanje potrebuje dva procesa: prvega za strežnik NodeJS, drugega za podatkovno bazo MySQL. Ustvariti bomo morali torej dva unikernela, za vsak proces enega. Ker se ne želimo ukvarjati z dvema različnima tipoma unikernelov, se odločimo, da bosta oba unikernela istega tipa. Povedano drugače, iščemo takšno implementacijo unikernela, ki bo omogočala:

1. poganjanje strežnika NodeJS
2. poganjanje podatkovne baze MySQL
3. poganjanje na hipervizorju KVM

Podpora za poganjanje na hipervizorju KVM je potrebni pogoj zaradi uporabe omenjene tehnologije pri postavitvi testnega oblačnega ogrodja OpenStack.

Tabela 4.1: Seznam odprtokodnih implementacij unikernelov. Pripisana so podprta programska okolja ('splošno' pomeni, da v unikernelu lahko poganjamo program, napisan v poljubnem programskem jeziku). V zadnjem stolpcu vrednost 'direktno' pomeni, da je unikernel možno poganjati neposredno na strojni opremi (in posledničnо s popolno virtualizacijo na poljubnem hipervizorju). Vrednost 'Xen' oz. 'KVM' pomeni, da unikernel podpira uporabo paravirtualizacije tega hipervizorja.
Opomba: v tabelo smo vključili samo implementacije, ki so odprtokodne in razvite vsaj do te mere, da jih je možno poganjati na hipervizorju.

UNIKERNEL	VRSTA APLIKACIJE	ZA HIPERVIZOR
HaLVM	Haskell	Xen
LING	Erlang	Xen
MirageOS	OCaml	Xen
ClickOS	specializiran za usmerjevalnike	Xen
HermitCore	C, C++, Fortran, Go	direktno
Runtime.js	JavaScript	KVM
IncludeOS	C++	KVM
Rumprun	splošno	Xen, KVM, direktno
OSv	splošno	Xen, KVM, direktno

Tabela 4.1 prikazuje seznam obstoječih odprtokodnih implementacij unikernelov[4]. V tabeli smo za vsak tip unikernela navedli vrsto aplikacije (programski jezik), ki jo lahko poganjamo v njem. Opazimo, da so nekateri tipi unikernelov namenjeni poganjanju aplikacij, napisanih v specifičnih jezikih, kot sta na primer Haskell ali Erlang. Takšni unikerneli za poganjanje naše spletne aplikacije seveda niso primerni, saj bi za njihovo uporabo morali ponovno implementirati aplikacijo Medo z uporabo ustreznega programskega jezika. Na seznamu opazimo tudi unikernel ClickOS, ki ni le omejen na en sam programski jezik (C++), temveč tudi na eno samo aplikacijo (visokozmogljivostni usmerjevalnik), kar ga naredi popolnoma neprimernega za poganjanje naše aplikacije. Nekoliko primernejši se zdi unikernel Runtime.js, ki je specializiran za poganjanje aplikacij napisanih v programskem jeziku JavaScript. Vendar smo ga ravno zaradi omejitve izključno na programski jezik JavaScript izločili iz seznama unikernelov za nadaljnjo obravnavo, saj ne izpolnjuje kriterija 2, ki pravi, da mora biti na izbranem tipu unikernela poleg aplikacije NodeJS možno poganjati tudi podatkovno bazo MySQL. Za naše potrebe se zdita primerna le dva tipa unikernela s tega seznama, ki edina izpolnjujeta vse tri pogoje: **Rumprun** in **OSv**.

V naslednjem poglavju se osredotočimo zgolj na omenjeni dve implementaciji. Pri posamezni implementaciji poskusimo zgraditi unikernele, potrebne za poganjanje testne aplikacije Medo, pri čemer ocenimo uporabniško izkušnjo postopka.

[4]Seznam smo zgradili na podlagi http://unikernel.org/projects

Poglavje 5

Uporabniška izkušnja pri uporabi unikernelov

Postavimo se v vlogo razvijalca spletnih aplikacij in se vprašajmo, s katerimi programskimi jeziki se ukvarja v svojem vsakdanu, kaj ima tipično nameščeno na svojem razvojnem računalniku. Verjetno si je predvsem domač z ogrodji za izdelavo spletnih aplikacij, kot sta na primer NodeJS, ki je napisan v programskem jeziku JavaScript, ali pa morda Django, ki je napisan v programskem jeziku Python. Dobro pozna tudi druge spletne tehnologije in jezike, ki se uporabljajo za prikaz spletnih strani (npr. HTML in CSS). Pri svojem delu se osredotoča na *uporabo* vseh teh tehnologij in se ne obremenjuje s tem, kako so narejene, katere sistemske klice uporabljajo.

Ko razvijalec konča z lokalnim razvojem spletne aplikacije, pogosto kreira nov splošnonamenski virtualni stroj v oblačni infrastrukturi in vanj namesti potrebno programsko opremo za poganjanje aplikacije. Pri tem se večino časa ukvarja s tem, kako namestiti aplikacijo, da bo delovala. Nikakor se ne loti prevajanja operacijskega sistema iz izvorne kode ali česa podobnega, ustreznih znanj za to praviloma nima.

Sedaj pa si predstavljamo zadrego, v kateri se znajde spletni razvijalec, če se odloči namesto splošnonamenskih virtualnih strojev uporabiti unikernel. V poglavju 3.3 smo omenili, da je osnovna ideja tehnologije unikernel ta,

da na navidezni strojni opremi neposredno poganjamo aplikacijo. Pri tem je potrebni nabor funkcionalnosti operacijskega sistema vgrajen v aplikacijo samo. To pomeni, da mora spletni programer svojo aplikacijo prevesti s posebnim namenskim prevajalnikom, specifičnim za izbrano implementacijo unikernela, ki poleg aplikacijske kode prevede tudi potrebne funkcionalnosti operacijskega sistema. Povedano drugače, spletni razvijalec se je prisiljen ukvarjati s kompleksno konfiguracijo prevajanja operacijskega sistema, za kar nima ustreznih znanj.

5.1 Kriteriji za ocenjevanje uporabniške izkušnje

Definirajmo lastne kriterije za ocenjevanje uporabniške izkušnje postopka priprave unikernela za poganjanje spletne aplikacije v oblačnem ogodju OpenStack. Kriteriji so kvalitativni in do neke mere subjektivni, a kljub temu v pomoč pri vrednotenju uporabniške izkušnje.

1. **Kompleksnost namestitve orodij**
   Različne implementacije unikernelov uporabljajo različna orodja za ustvarjanje in njihovo upravljanje. Namestitev teh orodij naj bo čim bolj enostavna. Ta metrika ima dve možni vrednosti: *enostavna* in *kompleksna*. Slednja pomeni, da je orodje potrebno prevajati iz izvorne kode, medtem ko *enostavna* pomeni, da obstaja vnaprej prevedena izvedljiva datoteka, ki jo je enostavno namestiti.

2. **Potreba po dodatnem znanju**
   Pri izgradnji unikernela uporabnika ne želimo obremeniti s kopico dodatnih informacij. Podrobnosti implementacije unikernela naj bodo skrite pred njim. Orodje, s katerim gradi unikernel, naj predstavlja visok nivo abstrakcije in naj od uporabnika zahteva le konfiguracijo, povezano s samo aplikacijsko kodo.

   Potrebo po dodatnem znanju na podlagi lastne presoje ocenimo bodisi z *majhna*, *srednja* ali *velika*, pri čemer je najbolj zaželjena vrednost

majhna.

3. **Prevajanje tuje izvorne kode**
   Prevajanje izvorne kode, ki ni del naše aplikacije, je nehvaležno opravilo, katerega uspešnost zavisi od združljivosti uporabnikovega razvojnega okolja z okoljem, kot ga predpostavljajo skripte za prevajanje. V kolikor pride do združljivostnih težav, se postopek prevajanja zaplete in potrebno je podrobno preučiti predpostavke tuje izvorne kode ter prilagoditi lastno razvojno okolje.

   Pri izgradnji unikernela naj uporabniku ne bo potrebno prevajati nobene druge izvorne kode razen izvorno kodo lastne aplikacije. Metrika ima dve možni vrednosti, *da* in *ne*, pri čemer je zaželjena vrednost *ne*.

4. **Možnost izbire velikosti in vsebine unikernela**
   Pri izgradnji unikernela je uporabniku smiselno ponuditi možnost vključitve poljubne kombinacije osnovnih in dodatnih funkcionalnosti ter izbiro želene velikosti ciljnega unikernela. Metrika ima dve možni vrednosti. *Popolna prilagodljivost* pomeni, da je možno izbrati poljubno kombinacijo funkcionalnosti in določiti poljubno velikost unikernela. *Delna prilagodljivost* pa pomeni, da je možno izbirati zgolj med množico vnaprej pripravljenih kombinacij funkcionalnosti, kjer vsaka kombinacija fiksira velikost unikernela. Zaželjena je *popolna prilagodljivost*, saj le tako lahko uporabnik zgradi unikernel optimalen za svoje potrebe.

5. **Čas izgradnje unikernela**
   Približno izmerimo čas izgradnje unikernela za našo testno spletno aplikacijo Medo. Izmerimo koliko časa preteče od zagona orodja za izgradnjo unikernela do konca izvajanja. Rezultat podamo z eno izmed naslednjih opisnih vrednosti: *nekaj sekund*, *nekaj deset sekund*, *nekaj minut*.

6. **Integracija z ogrodjem OpenStack**
   Omenili smo, da želi uporabnik pognati unikernel na oblačnem ogrodju

OpenStack. Ta metrika je namenjena ocenjevanju količine ročnega dela, ki ga mora uporabnik opraviti, da se zgrajeni unikernel na njem zažene. Možni sta dve vrednosti metrike: *nepodprto* pomeni, da orodje za izgradnjo unikernela ni integrirano z ogrodjem OpenStack. Uporabnik mora lastnoročno pretvoriti zgrajeni unikernel v primerno obliko in ga naložiti na ogrodje OpenStack ter ga tam pognati. *Podprto* pa pomeni, da so omenjeni koraki avtomatizirani. Zaželjena je vrednost *podprto*.

7. **Možnost lokalnega poganjanja unikernela**
   Ko uporabnik zgradi unikernel, mu je smiselno ponuditi možnost, da lokalno preveri, če deluje. To pomeni, da mora orodje nuditi možnost zagona unikernela in uporabniku pokazati izpis konzole aplikacije, ki teče v unikernelu. Metrika loči dve vrednosti: *uporabno* in *neuporabno*. Slednja vrednost pomeni, da lokalno poganjanje bodisi ni podprto, bodisi je implementirano na način, ki je za uporabnika neuporaben. Zaželjena je vrednost *uporabno*.

V nadaljevanju si ogledamo nekaj implementacij unikernelov in preverimo, kakšno uporabniško izkušnjo nudijo.

5.2 Ovrednotenje uporabniške izkušnje pred nadgradnjo

V nadaljevanju se osredotočimo na postopek poganjanja testne aplikacije Medo na izbranih dveh tipih unikernelov, Rumprun in OSv, in smo posebej pozorni na uporabniško izkušnjo pri postopku priprave posameznega tipa. Uporabniško izkušnjo ocenimo na podlagi lastnih kriterijev, definiranih v razdelku 5.1, ki v središče postavijo vidik spletnega programerja. Izkaže se namreč, da je izgradnja unikernela v splošnem kompleksen problem, ki povzroča nemalo frustracij uporabniku. S pomočjo omenjenih kriterijev identificiramo problematične značilnosti orodij za izgradnjo unikernelov. Ugotovimo, da je orodje za izgradnjo unikernela tipa Rumprun za uporabnika

neuporabno, saj nam kljub naporom ni uspelo pognati niti aplikacije NodeJS niti podatkovne baze MySQL. Popolnoma drugačno uporabniško izkušnjo pa doživimo pri unikernelu tipa OSv, ki zaradi posrečene kombinacije lastnosti implementacije omogoča enostavnejšo pripravo unikernela.

5.2.1 Uporabniška izkušnja pri Rumprun

V razdelku 3.3.4 smo omenili, da unikernel Rumprun ustvarimo s pomočjo pomožnih programov in skript. Spletni programer, ki želi pognati aplikacijo NodeJS v unikernelu Rumprun, mora najprej prenesti izvorno kodo osnovne implementacije Rumprun s portala GitHub, in jo prevesti ter namestiti s pomočjo posebne skripte. Namestitev traja nekaj minut in ni trivialna, saj je ob koncu potrebno ročno posredovanje. Programer si mora zapomniti vrednosti nekaterih internih spremenljivk in vedeti, katere skripte je potrebno izpostaviti v sistemsko pot. Sledi prenos razširitvenega modula s podporo za izvajalno okolje NodeJS (oz. MySQL) z ločenega repozitorija na portalu GitHub. Programer mora v ustrezni podmapi uporabiti program za avtomatsko prevajanje `make`, ki v desetih minutah proizvede izvedljivo datoteko s statično prevedenim okoljem NodeJS. Dobljeno datoteko potem skupaj z datotekami aplikacije Medo zapakiramo v binarno datoteko ki jo lahko lokalno poženemo z uporabo posebne skripte. Skripta uporabi hipervizor QEMU, ki zna neposredno pognati binarno datoteko, pri čemer sistemski posnetek (dejanski unikernel) ustvari zgolj interno.

Podoben je postopek ustvarjanja unikernela s podatkovnim strežnikom MySQL. Programer mora v ustrezni podmapi pognati program `make`, ki mu v desetih minutah ustvari izvedljivo datoteko s statično prevedenim strežnikom. Korak je bil v našem primeru kompleksen, saj je bilo potrebno popravljati zelo specifične dele konfiguracijske datoteke `Makefile`, da smo omogočili uspešno prevajanje na našem sistemu. Zadnji korak postopka je, podobno kot pri izgradnji unikernela z aplikacijo NodeJS, pakiranje dobljene datoteke v binarno datoteko in nato uporaba pomožne skripte za njen lokalni zagon.

Poganjanje unikernela Rumprun na ogrodju OpenStack ni podprto s

skriptami. Obstaja teoretična možnost, da z zgoraj opisanim postopkom zgradimo binarno datoteko in jo ročno pretvorimo v unikernel. Povedano drugače, ročno jo moramo ustvariti v sistemski posnetek, vanj prenesti omenjeno binarno datoteko in nastaviti ustrezen zagonski ukaz. Unikenel lahko nato, seveda zopet lastnoročno, prenesemo na ogrodje OpenStack in ga tam poženemo. Vendar je opisani postopek kompleksen in zahteva poglobljeno znanje o ustvarnju sistemskih posnetkov, ki ga spletni programer praviloma nima.

Tabela 5.1 prikazuje oceno uporabniške izkušnje pri uporabi unikernela Rumprun za poganjanje aplikacije na platformi OpenStack. Uporabniška izkušnja je bila po vseh kriterijih negativna. Že sama namestitev orodja je kompleksna in zahteva zapleteno prevajanje izvorne kode orodja. Po namestitvi orodja smo bili prisiljeni še v prevajanje tuje izvorne kode, in sicer izvajalnega okolja NodeJS in strežnika MySQL. Kot zanimivost omenimo, da je bila poraba sistemskih virov v primeru prevajanja izvajalnega okolja NodeJS absurdna, saj smo potrebovali kar 18 GB razpoložljivega delovnega pomnilnika. Ker je to preseglo zmogljivosti naše strojne opreme, smo morali pridobiti dodatno znanje o tem, kako povečati količino trdega diska, ki se sme uporabiti za potrebe delovnega pomnilnika (ang. swap space), kar je še dodatno negativno vplivalo na našo uporabniško izkušnjo ob pripravi unikernela. Obenem smo naleteli na pomanjkanje dokumentacije, ki nas je kot uporabnika puščalo v popolni nevednosti, kaj katere skripte sploh počno in katere argumente potrebujejo. Na primer, po uspešnem desetminutnem prevajanju izvorne kode izvajalnega okolja NodeJS nismo vedeli, kakšen rezultat smo sploh dobili in kaj naj z njim počnemo. S poglobljeno študijo strukture projekta in skript smo komaj uspeli slediti skopim korakom navodil za pripravo unikernela za poganjanje aplikacije NodeJS. Pri tem smo dobili unikernel velik ravno dovolj, da sta v njem lahko obstajala vsebina aplikacije Medo in izvajalno okolje NodeJS. Če bi naša aplikacija med svojim delovanjem potrebovala več prostora, bi se zagotovo znašli v težavah, saj nismo našli nobene možnosti za izbiro velikosti ciljnega unikernela. Prav tako

Tabela 5.1: Ocena uporabniške izkušnje pri uporabi unikernela Rumprun za poganjanje aplikacije na platformi OpenStack. Rdeče označena polja pomenijo negativen vpliv na uporabniško izkušnjo.

#	METRIKA	VREDNOST
1	Kompleksnost namestitve orodij	kompleksna
2	Potreba po dodatnem znanju	velika
3	Prevajanje tuje izvorne kode	da
4	Možnost izbire velikosti in vsebine unikernela	delna prilagodljivost
5	Čas izgradnje unikernela	nekaj minut
6	Integracija z ogrodjem OpenStack	nepodprto
7	Možnost lokalnega poganjanja unikernela	neuporabno

nismo zasledili možnosti posodabljanja vsebine zgrajenega unikernela. Lokalno nam je unikernel uspelo pognati, vendar le za trenutek, saj se je takoj zatem, ko je aplikacija začela teči, zgodila napaka. Ker je bil opis napake dolg, pognani unikernel pa je bil sposoben prikazati le zadnjih 23 vrstic izpisa, nismo nikoli izvedeli, do kakšne napake je pravzaprav prišlo, kar vsekakor ni bilo prijetno.

Uporabniško izkušnjo pri izgradnji unikernela tipa Rumprun zato ocenjujemo kot negativno. Posledica slabe izkušnje je, da nam ni uspelo v celoti zgraditi nobenega izmed dveh unikernelov, potrebnih za poganjanje testne spletne aplikacije, čeprav smo temu posvetili veliko količino časa. Bralec se na tem mestu morda vpraša, če je tehnologija unikernelov že sploh dovolj zrela, da jo lahko uporabljajo končni uporabniki. Odgovor je: da, vendar ne vse implementacije. Rumprun vsekakor še ni. Izkazalo se je namreč, da je izgradnja unikernelov tipa OSv, ki jo opišemo v naslednjem poglavju, občutno enostavnejša in prijaznejša do uporabnika kot izgradnja unikernelov tipa Rumprun.

5.2.2 Uporabniška izkušnja pri OSv

V razdelku 3.3.4 smo omenili, da OSv od drugih implementacij tehnologije unikernel izstopa zaradi uporabe dinamičnega povezovalnika, ki omogoča vnaprejšnje prevajanje funkcij operacijskega sistema in poljubnih delov aplikacij. V programskem jeziku Go [26] napisano orodje Capstan (ki so ga implementirali avtorji unikernela OSv za pomoč pri upravljanju unikernela) izkoristi to prednost in uporabniku ponudi katalog vnaprej pripravljenih unikernelov s prednameščenimi aplikacijami oz. izvajalnimi okolji, kar poenostavi poganjanje uporabnikove aplikacijske logike. Oglejmo si postopek izgradnje unikernela tipa OSv s pomočjo orodja Capstan.

Spletni programer, ki želi pognati aplikacijo NodeJS v unikernelu tipa OSv, mora najprej s spleta prenesti program Capstan. Programa ni potrebno prevajati ali nameščati, le shraniti ga je potrebno na disk. Programer mora nato v mapo, kjer se nahaja njegova aplikacija NodeJS, dodati konfiguracijsko datoteko in v njej navesti

1. ime unikernela iz kataloga,
2. seznam datotek svoje aplikacije, ki jih želi vključiti v unikernel, ter
3. začetni ukaz, ki naj se izvede ob zagonu unikernela.

Sledi uporaba orodja Capstan s klicem preprostega ukaza `capstan run` in v nekaj sekundah je unikernel OSv z našo aplikacijo zgrajen in lokalno pognan s pomočjo hipervizorja QEMU.

Z orodjem Capstan zgrajeni unikernel ni nič drugega kot sistemski posnetek v formatu imenovanem QCOW2, ki ga zna pognati široka množica hipervizorjev. Brez dodatnega prilagajanja ga lahko naložimo na ogrodje OpenStack in ga poženemo. Povedano drugače, s pomočjo nadzorne plošče OpenStack Horizon ga kot vsak drug sistemski posnetek naložimo na komponento OpenStack Glance in uporabimo komponento OpenStack Nova za zagon instance.

Tabela 5.2 prikazuje oceno uporabniške izkušnje pri uporabi unikernela tipa OSv za poganjanje aplikacije na platformi OpenStack. Kompleksnost

Tabela 5.2: Ocena uporabniške izkušnje pri izgradnji unikernela OSv. Rdeče označena polja pomenijo negativen vpliv na uporabniško izkušnjo.

#	METRIKA	VREDNOST
1	Kompleksnost namestitve orodij	enostavna
2	Potreba po dodatnem znanju	srednja
3	Prevajanje tuje izvorne kode	ne
4	Možnost izbire velikosti in vsebine unikernela	delna prilagodljivost
5	Čas izgradnje unikernela	nekaj sekund
6	Integracija z ogrodjem OpenStack	nepodprto
7	Možnost lokalnega poganjanja unikernela	uporabno

namestitve orodja je majhna, saj le prenesemo binarno datoteko in jo shranimo na poljubno mesto. Z uporabo standardnih ukazov za nameščanje programske opreme (na našem sistemu je to pomenilo uporabo ukaza `apt-get`) namestimo še hipervizor QEMU, ki ga orodje Capstan zahteva za svoje delovanje. S tem je postopek namestitve zaključen, orodje je pripravljeno za uporabo. Pri namestitvi nismo prevajali nikakršne izvorne kode. Naslednji korak je priprava konfiguracijske datoteke v korenski mapi testne aplikacije Medo, kar zahteva delno poznavanje tehnologije OSv - navesti moramo namreč ustrezen (potencialno kompleksen) zagonski ukaz, ki naj se požene ob zagonu unikernela. To je razlog za oceno `srednja` pri postavki *2 Potreba po dodatnem znanju*. Idealno bi bilo namreč, da bi uporabnik v konfiguracijski datoteki navedel le tiste nastavitve, ki se neposredno tičejo njegove aplikacije (na primer pot do izhodiščne datoteke aplikacije Medo). Po končanem urejanju konfiguracijske datoteke z ukazom `capstan run` zgradimo in zaženemo unikernel. Postopek same izgradnje podrobno opišemo v poglavju 6, na tem mestu omenimo le, da nikdar ne gradimo novega unikernela, temveč le dodajamo datoteke v obstoječega (kontekstualizacija). Uporabi se torej ustrezni vnaprej pripravljeni unikernel, ki ga predhodno pretočimo iz oddaljenega

repozitorija. Takšen postopek nas po eni strani razbremeni vsakršnega prevajanja izvorne kode (pretočeni unikernel že vsebuje vnaprej prevedeno jedro OSv in vnaprej prevedeno izvajalno okolje NodeJS), po drugi strani pa nas močno omeji (pretočeni unikernel je fiksne velikosti 10 GB in vsebuje fiksen nabor vnaprej prevedenih aplikacij). Zato smo možnost izbire velikosti in vsebine unikernela ocenili z `delna prilagodljivost`. Izgradnja unikernela je hitra in traja le nekaj sekund, saj ne prevajamo nikakršne izvorne kode - razen naše aplikacijske kode je že vse prevedeno in prisotno v unikernelu. Pri izgradnji moramo le kontekstualizirati že zgrajeni unikernel z lastnimi aplikacijskimi datotekami. To pomeni, da se aplikacijske datoteke preprosto prenesejo z lokalnega diska v unikernel. Čas prenosa je sorazmeren številu datotek, zato se pri velikih projektih čas kontekstualizacije lahko poveča.

Orodje Capstan z uporabo hipervizorja QEMU lokalno požene dobljeni unikernel. Pri tem izkoristi tehnologijo vtičnikov, s pomočjo katere standardni vhod, izhod in napako unikernela preusmeri kar v terminal, kjer smo pognali ukaz `capstan run`. To pozitivno vpliva na uporabniško izkušnjo, saj uporabniku ponudi na ogled celotno zgodovino izpisov unikernela in ne le zadnjih nekaj vrstic, kot smo videli pri uporabi skript za izgradnjo unikernela tipa Rumprun. Zato smo v tabeli 5.2 postavko *7 Možnost lokalnega poganjanja* ocenili z `uporabno`.

Uporabniško izkušnjo pri izgradnji unikernela tipa OSv v splošnem ocenjujemo kot dobro, vendar ni brez pomanjkljivosti. Kot najbolj moteči izpostavimo neprilagodljivost velikosti unikernela in slabo prilagodljivost vsebine. Pri sicer do uporabnika prijaznem orodju Capstan nas je zmotila tudi kompleksna priprava konfiguracijske datoteke, ki zahteva poznavanje podrobnosti strukture unikernelov iz oddaljenega repozitorija.

5.3 Utemeljitev izbire OSv

Postavili smo se v vlogo spletnega programerja in poskusili pognati testno aplikacijo Medo z uporabo dveh popolnoma različnih implementacij tehnolo-

gije unikernel, ki sta izmed vseh implementacij edini izpolnjevali naše osnovne tri zahteve (razdelek 4.3).

Uporabniška izkušnja pri pripravi unikernela Rumprun je bila neprijetna, za kar je do neke mere kriva kompleksnost prevajanja tuje izvorne kode sama po sebi. Unikernela tipa Rumprun namreč ne moremo zgraditi drugače, kot da na poseben način prevedemo izvorno kodo aplikacije. Rezultat prevajanja mora biti ena sama binarna datoteka (ki vsebuje tako logiko aplikacije, kot tudi logiko poenostavljenega operacijskega sistema), saj Rumprun ne podpira dinamičnega povezovanja več binarnih datotek med izvajanjem.

Po drugi strani pa smo pri OSv odkrili, da odločitev avtorjev za **uporabo dinamičnega povezovalnika** omogoča bistveno boljšo uporabniško izkušnjo. Prevajanje celotnega unikernela z vsemi funkcijami operacijskega sistema vred v eno samo binarno datoteko namreč ni več potrebno, temveč lahko prevedemo vsako stvar posebej in dinamični povezovalnik jih bo znal pravilno povezati med seboj. To posledično pomeni, da uporabniku lahko vnaprej prevedemo poljubne aplikacije, vsako v svojo binarno datoteko (na primer v obliko deljenega objekta), ki jih bo moral le prenesti v unikernel, in že jih bo lahko uporabil.

To lastnost nekoliko nerodno izkorišča orodje Capstan, ki uporabniku vnaprej pripravi celotne unikernele, s čimer ga po nepotrebnem omeji. V naslednjem poglavju zato opišemo in implemetiramo nadgradnjo orodja Capstan s podporo za vnaprej prevedene aplikacijske pakete - module. Nadgrajeni Capstan v še večji meri izkoristi potencial, ki ga nudi uporaba dinamičnega povezovalnika, in sicer tako, da uporabniku omogoči izgradnjo novega unikernela, kot da bi sestavljal kocke Lego. Posamezna kocka predstavlja vnaprej prevedeno aplikacijo in končni rezultat (unikernel) je sestavljen iz poljubne kombinacije kock (aplikacijskih paketov).

Poglavje 6

Nadgradnja orodja Capstan

V prejšnjem poglavju smo se postavili v vlogo končnega uporabnika unikernela OSv in prepoznali pozitiven vpliv vnaprej pripravljenih unikernelov na uporabniško izkušnjo. Vendar smo kmalu naleteli na neprijetno pomanjkljivost: ne moremo jih poljubno spreminjati.

V nadaljevanju naslovimo ta problem z uvedbo vnaprej pripravljenih *aplikacijskih paketov* kot zamenjavo za trenutne vnaprej pripravljene *unikernele*. Osnovna ideja je, da vnaprej pripravimo pakete, ki jih uporabnik lahko poljubno vključi v svoj unikernel. Opišemo implementacijo nadgradnje in predstavimo uporabljeni mehanizem za zagotavljanje kakovosti implementirane nadgradnje. Nadaljujemo z razdelkom o opisu implementacije avtomatizacije poganjanja zgrajenih unikernelov na oblačnem ogrodju OpenStack, s katero odpravimo zamudno rutinsko ročno nalaganje in poganjanje preko spletnega vmesnika OpenStack Horizon.

6.1 Podpora za upravljanje z aplikacijskimi paketi

Omenili smo, da obstoječa verzija programa Capstan uporabniku ponudi repozitorij vnaprej zgrajenih unikernelov z vgrajeno podporo za različne aplikacije. Algoritem 1 opisuje potek kontekstualizacije vnaprej zgrajenega uni-

kernela z lastno aplikacijsko logiko. Uporabnik se odloči za uporabo enega od vnaprej pripravljenih unikernelov in poda seznam svojih aplikacijskih datotek, ki naj se dodatno naložijo vanj. Capstan iz repozitorija pretoči izbrani unikernel in ga lokalno požene tako, da v njem zažene poseben program `cpoiod`[1], ki preko protokola TCP na vratih 10000 komunicira s Capstanom. Capstan mu preko omenjenega komunikacijskega kanala pošlje aplikacijske datoteke, ki jih je naštel uporabnik. Nato Capstan zaustavi unikernel (ki sedaj vsebuje tudi uporabnikove datoteke) in ga shrani v lokalni repozitorij.

V naslednjem koraku Capstan skopira unikernel iz lokalnega repozitorija in kopiji nastavi poljuben zagonski ukaz (ang. boot command), kot ga uporabnik navede v konfiguracijski datoteki. Tako konfiguriran oz. bolje rečeno kontekstualiziran unikernel, ki vsebuje uporabnikove datoteke in ima nastavljen zagonski ukaz po želji uporabnika, je končni rezultat - unikernel z uporabnikovo aplikacijo. Lahko ga lokalno poženemo s Capstanom ali pa ročno naložimo na oblačno ogrodje OpenStack in poženemo tam.

Algoritem 1 Psevdokoda kontekstualizacije unikernela (pred nadgradnjo)

1: $p, u, \vec{l} \leftarrow$ preberi konfiguracijo uporabnika
2: **if** unikernel z imenom p ni v lokalnem repozitoriju **then**
3: pretoči unikernel z imenom p iz oddaljenega repozitorija
4: **end if**
5: $P \leftarrow$ unikernel z imenom p v lokalnem repozitoriju
6: nastavi zagonski ukaz od P na `/tools/cpiod.so`
7: lokalno poženi unikernel P
8: naloži aplikacijske datoteke $\vec{l}$ v unikernel P preko vtičnika `localhost:10000`
9: zaustavi P
10: nastavi zagonski ukaz od P na u
11: // poženi kontekstualiziran unikernel P

Oglejmo si algoritem 1 po korakih. V prvem koraku Capstan iz konfiguracijske datoteke prebere ime obstoječega unikernela, ki ga želi uporabnik kontekstualizirati (spremenljivka `p`), zagonski ukaz, s katerim uporabnik želi po kontekstualizaciji zagnati unikernel (spremenljivka `u`), in seznam datotek,

[1] https://en.wikipedia.org/wiki/Cpio

ki jih uporabnik želi vključiti v unikernel tekom kontekstualizacije (spremenljivka $\vec{l}$). V korakih 2 do 4 Capstan po potrebi pretoči unikernel z imenom `p` iz oddaljenega repozitorija na lokalni disk uporabnika. Oddaljeni repozitorij ni nič drugega kot spletni strežnik (konkretno gre za strežnik v gostovanju pri ponudniku Amazon AWS[2]), na katerega je avtor orodja Capstan shranil nabor vnaprej pripravljenih unikernelov, vsakega z drugačno vsebino in unikatnim imenom[3]. Izraz *lokalni repozitorij* pomeni mapo na lokalnem disku uporabnika, kamor se shranijo pretočeni unikerneli[4]. Koraki 6 do 9 prikazujejo uporabo posebnega mehanizma, imenovanega **cpiod**, s katerim orodje Capstan z lokalnega diska uporabnika prenese poljubne aplikacijske datoteke v datotečni sistem unikernela. Mehanizem je revolucionaren, saj se ne ukvarja z neposrednim poseganjem v datotečni sistem, temveč zažene unikernel in mu posreduje datoteke preko vtičnika. Ta si sam shrani prejete datoteke na ustrezno mesto lastnega datotečnega sistema, nato ga zaustavimo. V koraku 10 mu nastavimo končni zagonski ukaz, ki bo ob naslednjem zagonu unikernela pognal uporabnikovo aplikacijo.

Z algoritmom 1 opisan postopek omeji uporabnika na uporabo omejenega nabora vnaprej pripravljenih unikernelov. Prvič, ti unikerneli imajo že ustvarjen datotečni sistem in sicer formatiran na fiksno velikost 10 GB. Velikosti datotečnega sistema naknadno ne moremo spreminjati, saj bi pri tem izgubili vse datoteke. Na tem mestu poudarimo, da velikost datotečnega sistema ne sovpada nujno z velikostjo, ki jo na disku dejansko zasede unikernel. Gre namreč za logično velikost, do katere se lahko unikernel razširi. Kljub temu je fiksna meja 10 GB moteča omejitev, ki ni vedno optimalna za poganjanje aplikacije. Drugič, unikerneli pripravljeni s strani avtorja orodja Capstan vsebujejo fiksno kombinacijo izvedljivih datotek, ki je naknadno ne moremo spreminjati. Uporabnik iz njih seveda ne more ustvariti poljubne kombinacije, zato je popolnoma odvisen od ponudbe v oddaljenem repozitoriju. Opisane omejitve naslovimo z nadgradnjo, predstavljeno v naslednjem

[2]na naslovu `https://s3.amazonaws.com/osv.capstan/`

[3]za poganjanje aplikacije Medo smo uporabili unikernel z imenom `coludius/osv-node`

[4]privzeto gre za mapo `$HOME/.capstan/repository`

razdelku.

6.1.1 Opis nadgradnje

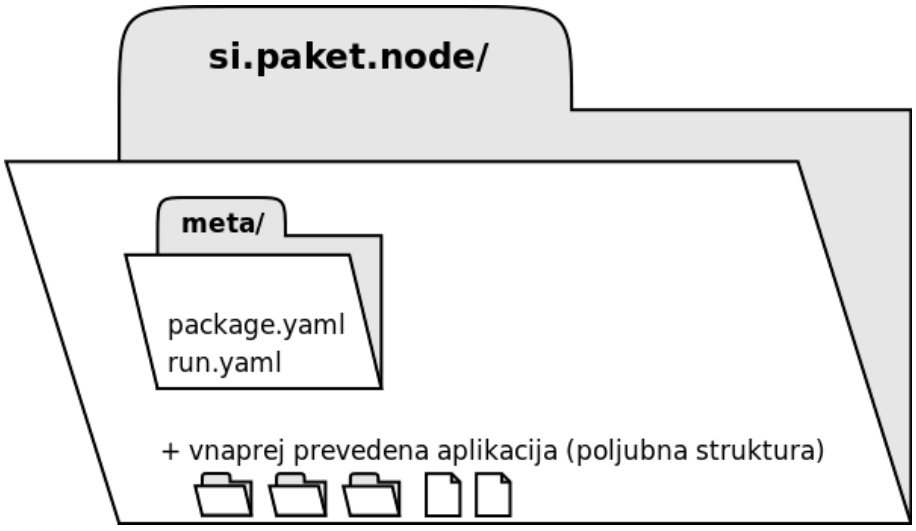
Izdelali smo naslednjo nadgradnjo orodja Capstan. Namesto vnaprej pripravljenih celotnih unikernelov oddaljeni repozitorij vsebuje raje vnaprej pripravljene module, tj. aplikacijske pakete, katerih strukturo definiramo v nadaljevanju. Uporabnik tako v konfiguracijski datoteki navede imena modulov, ki jih želi vključiti v svoj unikernel, in ne več imena unikernela. Capstan nato iz oddaljenega repozitorija pretoči izbrane module in iz njih zgradi nov unikernel poljubne velikosti.

Pozorni bralec se na tem mestu morda vpraša, v čem se nadgradnja sploh razlikuje od obstoječe rešitve. Odgovor je v zakasnitvi formatiranja datotečnega sistema. V obstoječi rešitvi je repozitorij vseboval celoten unikernel, torej sistemski posnetek s formatiranim datotečnim sistemom, na katerem so se nahajale vnaprej prevedene datoteke. Nova rešitev pa v repozitoriju ponuja le vnaprej prevedene datoteke, ki jih mora uporabnik šele prenesti na datotečni sistem unikernela. Korak formatiranja datotečnega sistema se torej prenese iz časa vnaprejšnje priprave unikernela (izvede ga avtor orodja Capstan) na čas kontekstualizacije unikernela (izvede ga končni uporabnik), kar da uporabniku svobodo pri izbiri velikosti datotečnega sistema.

Struktura modula

Z našo nadgradnjo smo uvedli novo entiteto, **modul**, ki ni nič drugega kot mapa z datotekami, kot prikazuje slika 6.1. Vsak modul mora vsebovati podmapo `meta`, kjer se nahajajo metapodatki o njem. Preostanek vsebine modula ni omejen z nobenimi specifikacijami in lahko vsebuje poljubno direktorijsko strukturo vnaprej prevedene aplikacije. Običajno modul vsebuje eno ali več datotek s končnico `.so`, v katerih se nahaja prevedena aplikacija ter eno ali več konfiguracijskih datotek v poljubni direktorijski strukturi.

Mapa `meta/` vsebuje dve datoteki. Datoteka **package.yaml** navaja splošne podatke o modulu, kot so ime, naziv in ime avtorja. Poleg tega vsebuje tudi



Slika 6.1: Struktura modula oz. aplikacijskega paketa. Vsak modul mora vsebovati mapo `meta/` z metapodatki, preostala struktura je poljubna. Vsebina modula se nespremenjena prenese v ciljni unikernel.

seznam imen modulov, ki so potrebni za delovanje. Ko v konfiguracijski datoteki zahtevamo nek modul, orodje Capstan v nastajajoči unikernel avtomatsko vključi celotno drevesno strukturo odvisnih modulov tega modula. Datoteka **run.yaml** pa vsebuje konkretne podatke o tem, kako pravilno poženemo modul. Bralec si datoteko run.yaml, ki jo podrobno opišemo v nadaljevanju, lahko predstavlja kot navodila za avtomatsko izgradnjo zagonskega ukaza za zagon modula.

Avtomatsko generiranje zagonskega ukaza

Končni uporabnik izmed vseh modulov izbere poljubnega oz. kombinacijo več modulov. To stori tako, da v konfiguracijski datoteki navede enolična imena modulov in Capstan jih avtomatsko pretoči iz oddaljenega repozitorija ter vključi v unikernel. V konfiguracijski datoteki pa mora uporabnik navesti tudi zagonski ukaz, ki naj se požene ob zagonu unikernela. To predstavlja potencialno zadrego, saj bi moral uporabnik poznati strukturo modula

in vedeti, kakšen (kompleksen) zagonski ukaz je potreben za njegov zagon. Oglejmo si primer. V okviru magistrskega dela smo pripravili modul, ki vsebuje vnaprej prevedeno izvajalno okolje NodeJS in ga shranili v oddaljeni repozitorij. Recimo, da ga želimo uporabiti za izvajanje naše testne aplikacije Medo (glej poglavje 4.1). Najprej moramo v konfiguracijski datoteki navesti njegovo enolično ime, da ga bo Capstan vključil v nastajajoči unikernel. Nato pa moramo v konfiguracijski datoteki navesti zagonski ukaz, ki izgleda natanko tako:

```
// Splosna oblika
--env={kljuc=vrednost} /usr/bin/libnode-4.4.5.so {vstopna-tocka}

// Konkreten primer za aplikacijo Medo
--env=PORT=3000 /usr/bin/libnode-4.4.5.so /bin/server.js
```

V prvem delu ukaza z uporabo sintakse, specifične za unikernel OSv, navedemo okoljske spremenljivke, ki naj se pojavijo v sistemskem okolju unikernela. V drugem delu ukaza navedemo pot do vnaprej prevedenega programa, ki naj se požene, torej `/usr/bin/libnode-4.4.5.so`. Ta datoteka je del modula z izvajalnim okoljem NodeJS in uporabnik se je praviloma ne zaveda. Tretji del ukaza sestavlja argument za program; konkretno gre za pot do vstopne datoteke naše aplikacije NodeJS.

Uporabnik lahko takšen zagonski ukaz sestavi na podlagi dokumentacije modula. Vendar je pri tem izpostavljen morebitni napaki in frustracijam zaradi nepoznavanja strukture modula, zato smo orodje Capstan nadgradili s podporo za avtomatsko generiranje zagonskega ukaza za izbrana izvajalna okolja, kot je na primer NodeJS. Pojem *izvajalno okolje* (ang. runtime) v kontekstu nadgradnje orodja Capstan s podporo za upravljanje z aplikacijskimi paketi (moduli) pomeni preprosto modul, ki ga uporabimo za izvajanje uporabnikove aplikacijske kode. Na primer, Aplikacija Medo je napisana v programskem jeziku JavaScript in za izvajanje potrebuje modul z izvajalnim okoljem NodeJS. Capstan smo nadgradili tako, da v konfiguracijski datoteki

uporabniku po novem ni več potrebno navesti celotnega zagonskega ukaza, temveč le vrednosti spremenljivk, ki so specifične za aplikacijo. Konfiguracija za zgornji primer se sedaj poenostavi v:

```
// Splosna oblika
runtime: {ime-izvajalnega-okolja}
... atributi specificni za izbrano izvajalno okolje

// Konkreten primer za aplikacijo Medo
runtime: node
main: /bin/server.js
env:
  PORT: 3000
```

V orodje Capstan smo torej vgradili logiko, ki na podlagi konfiguracijske datoteke zazna, da želimo pognati aplikacijo z uporabo izvajalnega okolja NodeJS. Capstan nato v unikernel avtomatsko vključi ustrezen modul, ki vsebuje izbrano izvajalno okolje, in uporabi podane atribute pri generiranju zagonskega ukaza. Opazimo, da uporabniku sedaj ni več potrebno navesti celotnega (kompleksnega) ukaza, temveč le manjkajoče parametre.

6.1.2 Implementacija nadgradnje

Podpora za aplikacijske pakete

Program Capstan smo opremili z novo logiko, ki je podprla zgoraj opisano nadgradnjo. Algoritem 2 prikazuje psevdokodo izgradnje (kontekstualizacije) unikernela ob uvedbi osnovne podpore za module. Podrobnosti implementacije so zaradi preglednosti izpuščene. Bistvena razlika med tem algoritmom in algoritmom 1 je v prvi vrstici. Pred nadgradnjo (algoritem 1) je moral uporabnik v konfiguracijski datoteki podati ime vnaprej pripravljenega unikernela p, ki že vsebuje fiksen nabor modulov in je fiksne velikosti. Ob uvedbi nadgradnje (algoritem 2) pa uporabnik poda poljuben seznam modulov, ki jih želi uporabiti, in poljubno velikost ciljnega unikernela.

Oglejmo si algoritem 2 po korakih. Orodje Capstan predpostavlja, da se pri izgradnji unikernela uporabnik nahaja v korenskem direktoriju aplikacije. V prvi vrstici (glej algoritem 2) preberemo konfiguracijsko datoteko, ki jo je pripravil uporabnik in vsebuje seznam imen modulov (spremenljivka $\vec{m}$), želeno velikost unikernela (spremenljivka `s`) in celoten zagonski ukaz (spremenljivka `u`). V drugi vrstici Capstan avtomatsko prebere vsebino trenutnega direktorija in si pridobi seznam poti do vseh datotek in map v njem. Sledi prenos baznega unikernela iz oddaljenega repozitorija (vrstice 3 do 6). Pojem **bazni unikernel** označuje poseben vnaprej pripravljen unikernel, ki se nahaja v oddaljenem repozitoriju in služi kot osnova za izgradnjo novega unikernela. Zgrajen je na poseben način. Konfiguriran je tako, da ob zagonu vsebino svojega datotečnega sistema shrani v delovni pomnilnik, formatira datotečni sistem in vsebino shrani nazaj na datotečni sistem. To nam omogoča, da v koraku 7 nastavimo poljubno logično velikost unikernela in ga nato poženemo (vrstice 8 do 10), s čimer se nastavljeni velikosti prilagodi tudi datotečni sistem. Naslednji korak je prenos modulov in aplikacijskih datotek v unikernel (vrstice 11 do 20). Tu nam je ponovno v pomoč pomožni mehanizem `cpiod`, ki smo ga opisali na začetku razdelka 6.1. Zadnji korak je nastavitev zagonskega ukaza, kot ga je predpisal uporabnik (vrstica 21). Tako pripravljen unikernel lahko brez nadaljnjih sprememb poženemo tudi brez uporabe orodja Capstan, saj je zagonski ukaz zapečen vanj.

Podpora za avtomatsko generiranje zagonskega ukaza

Potem, ko smo implementirali osnovno podporo za aplikacijske pakete (algoritem 2), smo implementirali še podporo za avtomatsko generiranje zagonskega ukaza za izbrana izvajalna okolja. Podprli smo dve izvajalni okolji, NodeJS in Javo. Nadgrajeno orodje Capstan v končni obliki sledi psevdokodi, ki je podana z algoritmom 3. Z vidika uporabnika in uporabniške izkušnje je pomembna predvsem prva vrstica algoritma, ki prikaže, da mora uporabnik v konfiguracijski datoteki podati le še nastavitve, ki so neposredno povezane z njegovo aplikacijo. Od njega se namreč ne zahteva več, da poda komple-

Algoritem 2 Psevdokoda kontekstualizacije unikernela (ob uvedbi podpore za aplikacijske pakete - module)

1: $\vec{m}, s, u \leftarrow$ preberi konfiguracijo uporabnika
2: $\vec{l} \leftarrow$ seznam datotek in direktorijev v trenutni mapi
3: **if** bazni unikernel ni v lokalnem repozitoriju **then**
4: pretoči bazni unikernel iz oddaljenega repozitorija
5: **end if**
6: $B \leftarrow$ bazni unikernel
7: povečaj velikost unikernela B na velikost s
8: nastavi zagonski ukaz od B na `/tools/mkfs.so`
9: lokalno poženi unikernel B (datotečni sistem se formatira)
10: zaustavi B
11: nastavi zagonski ukaz od B na `/tools/cpiod.so`
12: lokalno poženi unikernel B
13: **for all** m iz seznama izbranih modulov $\vec{m}$ **do**
14: **if** modul m ni v lokalnem repozitoriju **then**
15: pretoči modul m iz oddaljenega repozitorija
16: **end if**
17: naloži modul m v unikernel B preko vtičnika `localhost:10000`
18: **end for**
19: naloži aplikacijske datoteke $\vec{l}$ v unikernel B preko vtičnika `localhost:10000`
20: zaustavi B
21: nastavi zagonski ukaz od B na u
22: // poženi kontekstualiziran unikernel B

ksen zagonski ukaz, temveč le ime izvajalnega okolja (spremenljivka `i`) in aplikacijsko orientirane parametre zanj (spremenljivka $\vec{v}$).

Oglejmo si algoritem 3 korak za korakom. Program Capstan tudi tokrat predpostavlja, da se nahajamo v korenskem direktoriju aplikacije. V prvi vrstici preberemo konfiguracijsko datoteko, ki jo je pripravil uporabnik. Ta je enostavna in zahteva vnos zgolj takšnih parametrov, ki so povezani z našo aplikacijo: ime izvajalnega okolja (spremenljivka `i`), parametri izvajalnega okolja (spremenljivka $\vec{v}$) in velikost ciljnega unikernela (spremenljivka `s`). Na tem mestu omenimo, da uporabnik v konfiguraciji še vedno lahko poda seznam dodatnih modulov, ki naj se prenesejo v unikernel, čeprav za poganjanje same aplikacije to ni potrebno. V nadaljevanju algoritma namreč vidimo, da orodje Capstan glede na izbrano izvajalno okolje `i` avtomatsko vključi nujno potrebne module za poganjanje aplikacije (vrstica 13). Generiranje datotečnega sistema in prenos modulov ter aplikacijskih datotek v unikernel (vrstice 2 do 21) je enako kot pri alogritmu 2, z izjemo vrstice 13. Kot smo omenili, je generiranje seznama potrebnih modulov avtomatizirano, saj se Capstan zaveda, katero izvajalno okolje bomo poganjali. Zadnji korak ustvarjanja unikernela je nastavitev zagonskega ukaza, ki se avtomatsko zgenerira na podlagi parametrov $\vec{v}$ (vrstica 22). Nabor podprtih parametrov $\vec{v}$ se razlikuje glede na izbrano ime izvajalnega okolja i. Za `i=node` sta na primer na voljo dva parametra: pot do datoteke z vstopno točko aplikacije in seznam vrednosti (poljubnih) okoljskih spremenljivk.

6.2 Podpora za avtomatsko poganjanje unikernelov na ogrodju OpenStack

Orodje Capstan poenostavi izgradnjo unikernela za končnega uporabnika. Pri tem se rezultat, torej zgrajeni unikernel, shrani v lokalni repozitorij, do katerega uporabnik načeloma ne dostopa ročno. Lokalni repozitorij ni nič drugega kot mapa na datotečnem sistemu. V tej mapi se nahajajo unikerneli, datoteke s končnico `.qemu`, skupaj z metapodatki.

Algoritem 3 Psevdokoda kontekstualizacije unikernela (ob uvedbi podpore za avtomatsko generiranje zagonskega ukaza)

```
1: i, v⃗, s ← preberi konfiguracijo uporabnika
2: l⃗ ← seznam datotek in direktorijev v trenutni mapi
3: if bazni unikernel ni v lokalnem repozitoriju then
4:     pretoči bazni unikernel iz oddaljenega repozitorija
5: end if
6: B ← bazni unikernel
7: povečaj velikost unikernela B na velikost s
8: nastavi zagonski ukaz od B na /tools/mkfs.so
9: lokalno poženi unikernel B (datotečni sistem se formatira)
10: zaustavi B
11: nastavi zagonski ukaz od B na /tools/cpiod.so
12: lokalno poženi unikernel B
13: m⃗ ← seznam modulov potrebnih za izvajalno okolje i
14: for all m iz seznama modulov m⃗ do
15:     if modul m ni v lokalnem repozitoriju then
16:         pretoči modul m iz oddaljenega repozitorija
17:     end if
18:     naloži ekstrahiran modul m v unikernel B preko vtičnika localhost:10000
19: end for
20: naloži aplikacijske datoteke l⃗ v unikernel B preko vtičnika localhost:10000
21: zaustavi B
22: u ← generiraj zagonski ukaz za izvajalno okolje i na podlagi parametrov v⃗
23: nastavi zagonski ukaz od B na u
24: // poženi kontekstualiziran unikernel B
```

Capstan ponuja tudi ukaz za lokalno poganjanje unikernela s pomočjo hipervizorja QEMU[5]. Vendar v našem primeru ne želimo poganjati unikernela lokalno, temveč na oblačnem ogrodju OpenStack. Ker testna postavitev ogrodja OpenStack uporablja isti tip hipervizorja, kot ga uporabljamo za lokalno poganjanje, torej QEMU/KVM, je pravzaprav potrebno le prenesti zgrajeni unikernel na pravo mesto na ogrodju OpenStack in že ga lahko poženemo. Pozorni bralec bo na tem mestu verjetno uganil, na kateri komponenti je pravo mesto za zgrajeni unikernel: na komponenti za upravljanje s sistemskimi posnetki, imenovani OpenStack Glance. Uporabnik se prijavi v spletni vmesnik OpenStack Horizon, od koder iz lokalnega repozitorija orodja Capstan s pomočjo komponente OpenStack Glance uvozi zgrajeni unikernel. Zagon unikernela lahko prav tako sprožimo kar preko spletnega vmesnika OpenStack Horizon. Iz seznama sistemskih posnetkov, naloženih na komponento OpenStack Glance, izberemo naš unikernel in ga zaženemo z ustreznim odtenkom (ang. flavor). Uporabnik nato preko spletnega vmesnika OpenStack Horizon dodeli unikernelu navzven dostopen naslov IP ter mu omogoči promet preko ustreznih vrat. Na primer, za testno aplikacijo Medo moramo dovoliti omrežni promet TCP preko vrat, na katerih unikernel streže spletno aplikacijo.

6.2.1 Opis nadgradnje

Opazimo, da je postopek prenosa zgrajenega unikernela mogoče avtomatizirati. Zato smo orodje Capstan nadgradili z logiko, ki se poveže na vmesnik REST ogrodja OpenStack, preko katerega naloži zgrajeni unikernel v zbirko sistemskih posnetkov in ga požene.

Pri opisu uvedbe podpore za aplikacijske pakete v razdelku 6.1.2 smo omenili, da uporabnik v postopku kontekstualizacije lahko izbere poljubno velikost datotečnega sistema nastajajočega unikernela. Velikost je za zgrajeni

[5]Podprti so pravzaprav trije hipervizorji za lokalno poganjanje: QEMU, VirtualBox in VMware. Vendar smo uporabljali le QEMU, ki ga Capstan uporablja že pri sami izgradnji unikernela, in je zato namestitev neizogibna.

unikernel fiksna in je naknadno ni mogoče spreminjati, saj bi pri tem izgubili vse datoteke. Omejitev moramo upoštevati pri poganjanju unikernela na ogrodju OpenStack, kjer v koraku poganjanja instance izberemo odtenek. Če namreč zgradimo unikernel z datotečnim sistemom velikosti 100 MB, bo njegova velikost omejena na to vrednost. Tudi če pri poganjanju zanj izberemo odtenek z lastnostmi `1 CPU | 10 GB HDD | 512 MB RAM`, bo njegova velikost še vedno omejena na 100 MB in ne na 10 GB, kot bi morda pričakovali. Zadrego smo rešili z avtomatskim povečevanjem velikosti, ki jo poda uporabnik v konfiguraciji, na velikost najbližjega odtenka. Pri podani velikosti 100 MB bi orodje Capstan tako najprej poiskalo najmanjši odtenek, ki še lahko poganja takšno velikost. Recimo, da je to ravno odtenek naveden zgoraj, torej z lastnostjo `10 GB HDD`. Tedaj bo orodje Capstan v fazi kontekstualizacije unikernela pripravilo unikernel velikosti 10 GB in ne 100 MB, s čimer se izognemo nepotrebni neizkoriščenosti izbranega odtenka.

6.2.2 Implementacija nadgradnje

Algoritem 4 opisuje integracijo orodja Capstan z ogrodjem OpenStack. Gre za neposredno razširitev algoritma 3, pri čemer je implementirana dodatna logika za dinamično prilagajanje velikosti nastajajočega unikernela. Orodje Capstan iz okoljskih spremenljivk najprej zajame podatke za avtentikacijo uporabnika z ogrodjem OpenStack (korak 2). Okoljske spremenljivke nastavimo s pomočjo posebne skripte, ki jo prenesemo s klikom na gumb v spletnem vmesniku OpenStack Horizon. Vsebujejo pa podatke kot so uporabniško ime, geslo in enolični identifikator projekta v ogrodju OpenStack, na katerega želimo naložiti unikernel. Orodje Capstan nato na ogrodje OpenStack pošlje poizvedbo o obstoječih odtenkih (korak 3) in izmed vseh izbere tistega, ki je optimalen glede na izbrano velikost `s` (korak 4). V naslednjem koraku (korak 5) Capstan posodobi izbrano velikost `s` tako, da se ujema z velikostjo optimalnega odtenka. Nadaljnja izgradnja unikernela se izvede glede na novo vrednost `s`. Zgrajeni unikernel se nato avtomatsko naloži na ogrodje OpenStack (korak 7) in se tam tudi požene (korak 8). Pri tem je

zagotovljeno, da bo pognani unikernel lahko izkoristil ves diskovni prostor, ki mu ga dovoljuje odtenek.

Algoritem 4 Psevdokoda izgradnje unikernela z avtomatskim poganjanjem na oblačnem ogrodju OpenStack

1: $s, ... \leftarrow$ preberi konfiguracijo uporabnika
2: $\vec{a} \leftarrow$ zajemi podatke za avtentikacijo iz okoljskih spremenljivk
3: $\vec{f} \leftarrow$ pridobi seznam razpoložljivih odtenkov (uporabi $\vec{a}$ za dostop)
4: $f \leftarrow$ iz seznama $\vec{f}$ izberi optimalni odtenek za velikost s
5: $s \leftarrow$ velikost diska, kot jo specificira odtenek f
6: $U \leftarrow$ kontekstualiziraj unikernel, pri tem uporabi posodobljen s
7: naloži unikernel U na komponento Glance (uporabi $\vec{a}$ za dostop)
8: instanciraj unikernel U s komponento Nova, uporabi odtenek f (uporabi $\vec{a}$ za dostop)

6.3 Zagotavljanje kakovosti implementacije

Medsebojno pregledovanje programske kode (ang. peer code review) je pomemben del razvojnega procesa programske opreme, še posebno odprtokodne. Medsebojno pregledovanje kode pomeni, da programer implementira neko novo funkcionalnost in jo izpostavi kritiki drugega programerja [32]. Prvi programer nastopi v vlogi avtorja nove funkcionalnosti, drugi pa v vlogi njenega pregledovalca (ang. reviewer). Postopek implementacije posamezne funkcionalnosti postane iterativen. Avtor toliko časa odpravlja napake, ki jih odkrije pregledovalec, dokler bodisi pregledovalec ne izrazi pozitivne kritike bodisi avtor obupa in zavrže novo funkcionalnost. Šele ko je pregledovalec zadovoljen, se funkcionalnost sprejme v uradno verzijo programa (ang. upstream) in postopek je zaključen.

Študije [18, 1, 12] pokažejo, da s pomočjo medsebojnega pregledovanja kode bistveno zmanjšamo število nepravilnosti v kodi. Nekoliko presenetljivo pa Bacchelli et al. v [4] ugotovijo, da medsebojno pregledovanje kode poleg višje kakovosti izdelka prinese tudi druge prednosti, na primer prenos znanja, okrepitev odnosov med programerji in uporabo alternativnih pristopov

k reševanju problema. Kot zanimivost omenimo, da se medsebojno pregledovanje kode ne uporablja le pri programiranju odprotkodnih rešitev, temveč tudi komercialnih. V [5] avtorji analizirajo stanje v podjetju Microsoft in ugotovijo, da se medsebojno pregledovanje tam uporablja v tolikšni meri, kot pri razvoju odprtokodnih rešitev. Povprečno programer pregleduje kodo drugemu programerju 6 ur na teden, kar predstavlja 15 odstotkov njegovega delovnega časa.

Rezultat popularnosti medsebojnega pregledovanja kode, tako v odprtokodnem kot komercialnem segmentu programskega sveta, je obstoj kakovostnih orodij za pomoč pri tem opravilu. Gerrit je popularno orodje, ki olajša delo tako avtorju nove funkcionalnosti, ki z njegovo pomočjo dostavi kodo do pregledovalca, kot tudi pregledovalcu, ki z njegovo pomočjo komentira posamezne izseke kode.

6.3.1 Orodje Gerrit

Gerrit[6] je odprtokodni spletni strežnik, napisan v programskem jeziku Java, ki olajša komunikacijo med avtorjem in pregledovalcem pri medsebojnem pregledovanju kode. Morales et. al v [21] opišejo tipičen postopek uporabe orodja Gerrit. Avtor implementira novo funkcionalnost v obliki množice sprememb obstoječe kode (ang. patch set) in jo naloži na spletni strežnik Gerrit. Preko spletnega vmesnika nato povabi k sodelovanju enega ali več pregledovalcev. Pregledovalcem se v spletnem strežniku prikaže množica vseh sprememb in jim je ponujena možnost podajanja splošnega komentarja ter komentiranja posameznih datotek oz. posameznih vrstic datoteke. Avtor ima nato možnost odgovarjati na komentarje ter upoštevajoč komentarje posodabljati množico sprememb.

Pregledovalec na podlagi razprave in rezultata iterativnega posodabljanja kode s strani avtorja poda svojo oceno aktualne množice sprememb. Možne so štiri ocene: `-1` pomeni nestrinjanje s spremembo, `0` pomeni neopredeljenost, `1` pomeni strinjanje s spremembo in `2` pomeni absolutno strinjanje s

[6] https://github.com/gerrit-review/gerrit

spremembo. Ko se vsi pregledovalci strinjajo s spremembo, je ta sprejeta v uradno verzijo programa. Orodje Gerrit omogoča tudi popolno integracijo s sistemom za nadzor različic (ang. version control system, VCS), kar pomeni, da spremembo sprejmemo v uradno verzijo programa kar s klikom na gumb na spletnem vmesniku orodja Gerrit.

Orodje Gerrit smo uporabili tudi pri naši implementaciji nadgradenj in z njegovo pomočjo ter dragoceno pomočjo skupnosti, ki je nastopila v vlogi pregledovalca, zagotovili višjo kakovost implementacije. Postopek podrobneje opišemo v naslednjem razdelku.

6.3.2 Uporaba orodja Gerrit pri implementaciji

Nadgradnje orodja Capstan (glej razdelka 6.1.2 in 6.2.2) smo implementirali v majhnih korakih in uporabili medsebojno pregledovanje kode s pomočjo orodja Gerrit. Skupnost, ki v okviru evropskega projekta MIKELANGELO[7] skrbi za orodje Capstan, nam je pregledala vsako vrstico spremenjene kode. Komunikacija je potekala preko postavitve orodja Gerrit, ki ga je namestila in ustrezno konfigurirala skupnost.

Tipičen postopek implementacije posameznega koraka je bil sledeč. Implementirali smo majhno zaključeno celoto in vmesne varnostne kopije shranjevali v lasten repozitorij Git. Ko smo bili z implementacijo zadovoljni, smo vse lastne spremembe združili v eno samo uveljavitev sprememb (ang. commit), jo opremili z nazornim sporočilom, in potisnili na strežnik Gerrit. Skupnost je nato spremembe pregledala, jih testirala, in nam preko sistema Gerrit posredovala povratno informacijo. Na nas je bilo, da smo preučili dobljeno kritiko in dopolnili spremembe. Sledilo je odgovarjanje na komentarje skupnosti ter potisk novih sprememb na strežnik Gerrit. Skupnost je ponovno pogledala spremembe in podala komentarje.

Običajno smo potrebovali tri ali štiri iteracije, preden smo vključili spremembo v glavni program (Capstan). Začetna iteracija je tipično prejela dvajset komentarjev za posamezne izseke kode, nato vsaka iteracija manj.

[7] https://www.mikelangelo-project.eu

Med postopkom nismo odpravljali le napak na nivoju programske vrstice, temveč smo ponekod popolnoma spremenili pristop k rešitvi. Čeprav smo medsebojno pregledovanje kode sprva neradi uporabljali, saj smo se izpostavili skupnosti, se je to izkazalo za neupravičeno. Kot so v [4] avtorji pravilno navedli, se je skupnost potrudila za krepitev medprogramerskih odnosov in je bila pri podajanju kritike vljudna in potrpežljiva. Končni rezultat je tako kakovostna programska koda, ki pravilno deluje in jo je možno enostavno vzdrževati.

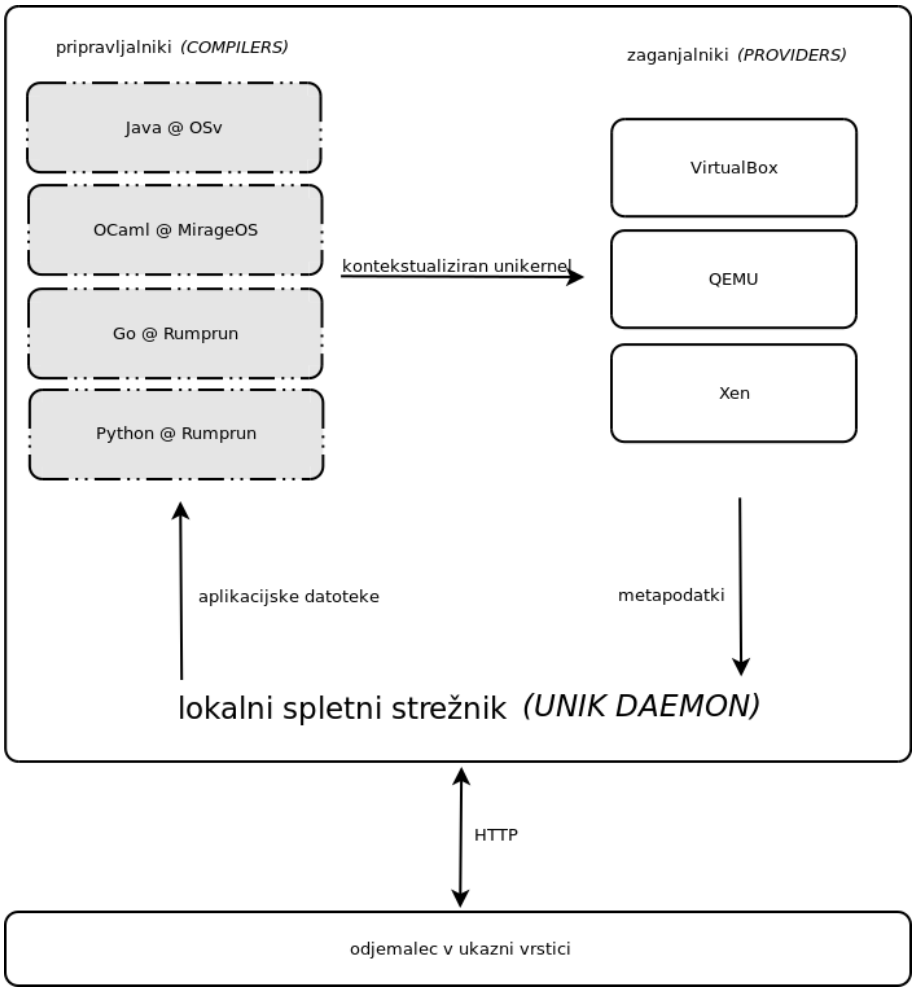
Poglavje 7

Integracija v platformo UniK

UniK[1] je popolnoma novo odprtokodno orodje, ki se je pojavilo tekom naše implementacije nadgradenj orodja Capstan. Namen orodja je izboljšanje uporabniške izkušnje pri poganjanju aplikacije z uporabo tehnologije unikernel. UniK naslavlja podoben problem, kot naše nadgradnje orodja Capstan, vendar na višjem nivoju abstrakcije. Medtem ko orodje Capstan ustvarja in poganja izključno unikernele tipa OSv, se orodje UniK ne omeji na specifično implementacijo unikernela. Podpira ustvarjanje in poganjanje unikernelov tipa Rumprun, IncludeOS, MirageOS in OSv. Orodje UniK samo po sebi ne implementira dejanske logike za ustvarjanje unikernelov, temveč za izgradnjo posameznega tipa uporabi obstoječe orodje oz. nabor pomožnih skript. Na tem mestu omenimo, da orodje UniK sicer uporabljamo iz ukazne vrstice, vendar se ukazi v ozadju prevedejo v zahtevke protokola HTTP in pošljejo na lokalni spletni strežnik imenovan *UniK Daemon*. Le-ta pa vsebuje logiko za ustvarjanje in poganjanje unikernelov.

Slika 7.1 prikazuje poenostavljeno shemo arhitekture orodja UniK. Opazimo, da je UniK zgrajen iz dveh delov. **Odjemalec v ukazni vrstici** pretvarja ukaze v zahtevke protokola HTTP in jih posreduje spletnemu strežniku. **Spletni strežnik** pa dejansko orkestrira ustvarjanje unikernelov. Za izgradnjo uporablja posebej pripravljene vsebnike Docker (zgoraj levo). Posame-

[1] https://github.com/emc-advanced-dev/unik



Slika 7.1: Visokonivjoska arhitektura orodja UniK.

zni vsebnik je specializiran za pakiranje specifične aplikacije v specifičen tip unikernela. Vsebnik, ki je na sliki 7.1 poimenovan z `OCaml @ MirageOS`, je na primer specializiran za izgradnjo unikernela tipa MirageOS za aplikacijo, napisano v programskem jeziku OCaml. Podobno je vsebnik `Go @ Rumprun` specializiran za izgradnjo unikernela tipa Rumprun za poganjanje aplikacij, napisanih v programskem jeziku Go. Zgrajeni unikernel se s pomočjo komponente, ki podpira ustreznega hipervizorja (zgoraj desno) nato tudi požene. Če je uporabnik zahteval zagon unikernela na primer na hipervizorju VirtualBox, se bo uporabila komponenta, ki podpira poganjanje na tem hipervizorju (na sliki je označena z `VirtualBox`).

UniK uporabimo tako, da se pomakenemo v korensko mapo naše aplikacije in tam ustvarimo konfiguracijsko datoteko. Vsebina datoteke zavisi od izbire tipa unikernela, programskega jezika naše aplikacije in izbranega hipervizorja. Iz ukazne vrstice sprožimo ustvarjanje in zagon unikernela. Aplikacijske datoteke se pošljejo lokalnemu spletnemu strežniku, ki jih shrani na primerno mesto. V naslednjem koraku spletni strežnik izmed množice vsebinkov Docker požene tistega, ki iz aplikacijskih datotek za izbrani programski jezik zgradi kontekstualiziran unikernel izbranega tipa. Unikernel nato prevzame tista komponenta za delo s hipervizorji, ki si jo je v konfiguracijski datoteki izbral uporabnik, in ga požene na hipervizorju. Cikel se zaključi s posredovanjem metapodatkov o zagnanen unikernelu odjemalcu v ukazno vrstico.

Arhitektura orodja UniK omogoča modularno dodajanje tako novih kombinacij tipov unikernela in programskega jezika, kot tudi dodajanje podpore novih hipervizorjev. V okviru magistrskega dela se nam je zdelo smiselno v čim večji meri vključiti nadgrajeno orodje Capstan v orodje UniK. Prvi korak je bila implementacija komponente za poganjanje na hipervizorju OpenStack (arhitekturno gledano gre za novo komponento zgoraj desno na sliki 7.1). Drugi korak pa je bila priprava vsebnika Docker, ki z uporabo nadgrajenega orodja Capstan zgradi unikernel tipa OSv (slika 7.1, levo zgoraj). Podroben opis implementacije omenjenih dveh komponent bomo zaradi preglednosti

izpustili, opisali bomo le uporabljeni način zagotavljanja kakovosti implementacije.

7.1 Zagotavljanje kakovosti implementacije

GitHub[2] je priljubljen portal za brezplačno shranjevanje odprtokodnih projektov [7]. Temelji na sistemu za nadzor različic imenovanem Git in poleg shranjevanja kode ponuja množico orodij za odprtokodni način programiranja. Eno izmed takšnih orodij oz. mehanizmov se imenuje *pull request* in je posredno namenjeno medsebojnem pregledovanju kode.

7.1.1 Mehanizem *pull request*

Oglejmo si tipično uporabo mehanizma *pull request*. Denimo, da se v nekem repozitoriju na portalu GitHub nahaja aplikacija, ki jo želimo nadgraditi. Imenujmo ta repozitorij *upstream*. Ker nismo lastnik aplikacije, nam ni dovoljeno neposredno spreminjati kode v repozitoriju *upstream*. Lahko pa shranimo identično kopijo repozitorija *upstream* v nov repozitorij. Lastnik nastalega repozitorija bomo seveda mi, zato bomo v ta repozitorij smeli neposredno vnašati svoje spremembe. V nekem trenutku bomo želeli svoje spremembe pokazati avtorju repozitorija *upstream* in mu predlagati, da jih vključi v svoj repozitorij. Tedaj bomo potrebovali mehanizem *pull request*, ki nam omogoča, da izberemo podmnožico sprememb v našem lastnem repozitoriju in jih pošljemo na ogled lastniku repozitorija *upstream*. Če so mu spremembe všeč, jih lahko s klikom na gumb sprejme v svoj repozitorij.

Vendar običajno spremembe niso sprejete brez popravkov. Lastnik repozitorija *upstream* tukaj prevzame vlogo pregledovalca kode, podobno kot smo videli že pri orodju Gerrit (glej poglavje 6.3.1). Pregledovalcu se prikaže seznam sprememb in lahko poda bodisi splošne komentarje, ali pa komentira posamezno vrstico kode. Avtor sprememb ima nato možnosti odgovoriti na komentarje in dopolnjevati svojo nadgradnjo. Postopek je iterativen: avtor

[2]https://github.com

svoje spremembe nadgrajuje toliko časa, dokler ni pregledovalec zadovoljen in jih sprejme v glavni program.

Mehanizem *pull request* ponuja podobno funkcionalnost kot orodje za medsebojno pregledovanje programske kode Gerrit. Ločita se predvsem po številu medsebojno odvisnih zaporednih sprememb, ki jih lahko hkrati izpostavimo pregledovalcu v pregled. Orodje Gerrit zahteva strnitev vseh sprememb, ki jih prinaša nova nadgradnja, v eno samo uveljavitev sprememb. Pri uporabi mehanizma *pull request* pa je dovoljeno izpostaviti poljubno število zaporednih uveljavitev sprememb naenkrat. Omenjena razlika je majhna, zato je uporaba enega ali drugega orodja običajno odvisna od osebne preference pregledovalcev kode. V primeru odprtokodnega projekta UniK so se avtorji odločili za uporabo mehanizma *pull request*, zato v naslednjem razdelku opišemo, kako smo slednjega uporabljali pri implementaciji.

7.1.2 Uporaba mehanizma *pull request* pri implementaciji

Nadgradnje orodja UniK smo implementirali po korakih in uporabili medsebojno pregledovanje kode s pomočjo mehanizma *pull request*. Skupnost, ki vzdržuje repozitorij z izvorno kodo programa UniK na portalu GitHub, je nastopila v vlogi pregledovalca in temeljito pregledala spremembe, preden jih je sprejela v svoj repozitorij.

Postopek implementacije je bil sledeč. Ustvarili smo identično kopijo repozitorija projekta UniK in v tako pridobljeni repozitorij tekom implementacije nadgradnje shranjevali varnostne kopije. Ko smo bili z implementacijo zadovoljni, smo uredili lastne spremembe v primerno zaporedje uveljavitev sprememb in vsako opremili z nazornim sporočilom. Nato smo s pomočjo spletnega vmesnika portala GitHub začeli postopek pregleda kode z uporabo mehanizma *pull request*. Skupnost je spremembe pregledala, jih testirala in nam podala kritiko. Na nas je bilo, da smo preučili dobljeno kritiko in dopolnili spremembe. Sledilo je odgovarjanje na komentarje skupnosti ter potisk novih sprememb na portal GitHub. Skupnost je nato ponovno pregledala

spremembe in podala nove komentarje. Postopek smo ponavljali, dokler niso bili naslovljeni vsi komentarji. Za posamezno spremembo smo potrebovali razpravo dolžine 30 do 50 komentarjev, preden jo je skupnost vključila v glavni repozitorij.

Poglavje 8

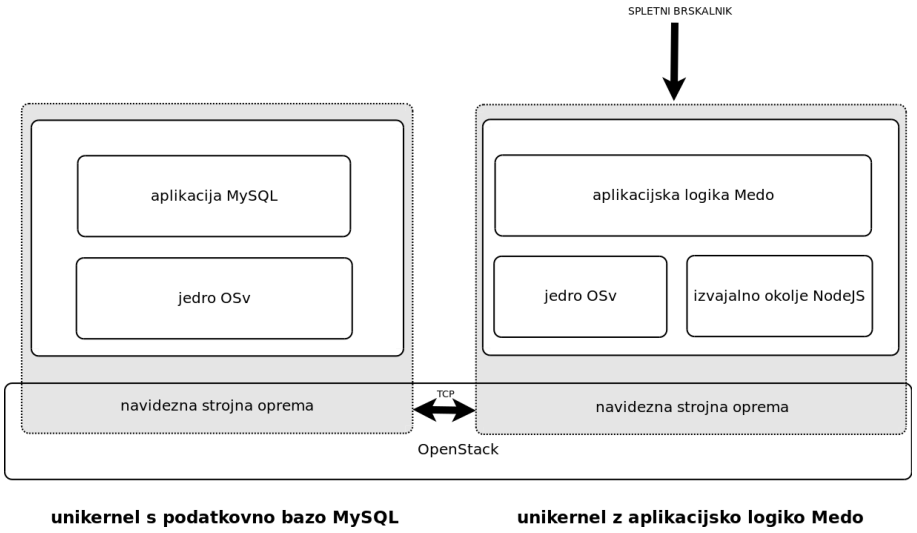
Ovrednotenje rezultatov

To poglavje je namenjeno demonstraciji učinka implementiranih nadgradenj orodja Capstan. Prikažemo, kako enostavno je po novem pognati aplikacijo Medo na ogrodju OpenStack z uporabo tehnologije unikernel. Omenili smo že, da moramo za uspešen zagon aplikacije Medo uporabiti dva unikernela: enega za poganjanje podatkovne baze MySQL in drugega za poganjanje aplikacijske logike v okolju NodeJS (glej sliko 8.1). V prvem primeru le zgradimo unikernel z obstoječim aplikacijskim paketom, ki vsebuje vnaprej prevedeno aplikacijo MySQL[1], in ga poženemo. V drugem primeru pa v unikernel vključimo tako obstoječ aplikacijski paket z vnaprej prevedenim izvajalnim okoljem NodeJS[2], kot tudi poslovno logiko naše testne aplikacije Medo. Za pravilen zagon drugega unikernela poskrbi v orodje Capstan vgrajena podpora za avtomatsko generiranje zagonskega ukaza za izvajalna okolja.

V nadaljevanju si podrobneje ogledamo postopek priprave posameznega unikernela in na koncu ocenimo uporabniško izkušnjo na podlagi lastnih kriterijev, ki smo jih definirali v razdelku 5.1. Predpostavimo, da je nadgrajeno orodje Capstan že nameščeno na razvojnem računalniku.

[1] aplikacijski paket MySQL smo pripravili v okviru magistrskega dela

[2] tudi aplikacijski paket NodeJS smo pripravili v okviru magistrskega dela



Slika 8.1: Aplikacijo Medo poženemo z uporabo dveh unikernelov. V prvem teče podatkovna baza MySQL (*levo*), v drugem pa strežnik NodeJS (*desno*).

8.1 Uporaba poljubnega aplikacijskega paketa

Oglejmo si postopek ustvarjanja in poganjanja unikernela s podatkovno bazo MySQL. Uporabnik najprej ustvari prazno mapo, poimenujmo jo `medo-db`, in v njej podmapo `meta` z dvema besedilnima datotekama `package.yaml` in `run.yaml`. V prvo vnese metapodatke o svoji aplikaciji, kot so ime aplikacije, ime avtorja in seznam modulov, ki naj se prenesejo v unikernel. Seznam modulov v našem primeru vsebuje ime enega samega modula, modula MySQL. Končna vsebina datoteke `package.yaml` je natanko takšna:

```
name: medo-db
title: Podatkovna baza aplikacije Medo
author: Miha Plesko
require:
  - si.paket.mysql-5.6.21
```

Druga datoteka, torej datoteka `run.yaml`, vsebuje podatke, potrebne za zagon unikernela. Ker želimo v našem primeru zagnati modul MySQL, vsebino

datoteke `run.yaml` nespremenjeno prenesemo iz istoimenske datoteke tega modula. S tem je priprava konfiguracijskih datotek zaključena.

Uporabnik nato odpre ukazno okno in se z uporabo ukaza `cd` pomakne v korensko mapo aplikacije (mapa `medo-db`) in tam požene ukaz:

```
$ capstan stack push --size 20GB
```

Zgornji ukaz pripravi unikernel (glej algoritem 3) in ga naloži na oblačno ogrodje OpenStack (glej algoritem 4). Uporabnik lahko nato s pomočjo brskalnika z nekaj kliki miške ročno zažene unikernel na na ogrodju OpenStack ali pa uporabi ukaz:

```
$ capstan stack run
```

in orodje Capstan v imenu uporabnika zažene unikernel. Čez nekaj trenutkov unikernel že teče in poganja podatkovno bazo MySQL. Iz dokumentacije modula MySQL razberemo privzeto uporabniško ime in geslo, ki ju lahko uporabimo za dostop do podatkovne baze. Privzeto je podatkovna baza prazna in vsebuje le prazno shemo z imenom `default`.

Kot zanimivost omenimo, da aplikacijski paket MySQL ob zagonu požene skripto s fiksne lokacije[3], ki nastavi privzeto uporabniško ime in geslo ter ustvari shemo z imenom `default`. To skripto lahko enostavno nadomestimo, saj se pri ustvarjanju unikernela vanj najprej prenesejo datoteke modulov, nato pa jih prepišejo datoteke aplikacije. Povedano drugače, če ustvarimo skripto na isti lokaciji relativno glede na korensko mapo aplikacije, kot že obstaja znotraj modula, bo ta prepisala skripto iz modula. Na ta način lahko nadzorujemo pripravo podatkovne baze (uporabniško ime, geslo, idr.) ob zagonu unikernela.

V nadaljevanju predpostavimo, da podatkovna baza teče v unikernelu na ogrodju OpenStack in da poznamo njen naslov IP.

[3]gre za lokacijo `/etc/mysql-init.sql` na datotečnem sistemu unikernela

8.2 Uporaba podprtega izvajalnega okolja

Oglejmo si še postopek ustvarjanja in poganjanja unikernela z aplikacijsko logiko NodeJS. Na tem mestu poudarimo, da enak rezultat lahko dosežemo na dva načina. Prvi način je kompleksnejši in zahteva podrobnejše poznavanje obstoječega modula s prevedenim izvajalnim okoljem NodeJS. Koraki pri tem načinu bi bili podobni kot pri izgradnji unikernela s podatkovno bazo MySQL, opisani v prejšnjem poglavju, le da uporabnik ne bi mogel uporabiti nespremenjene datoteke `run.yaml` iz modula NodeJS. Datoteko bi moral namreč prilagoditi svoji aplikacijski logiki. Drugi način pa je enostaven in izkoristi predstavljeno podporo za avtomatsko generiranje zagonskega ukaza (glej razdelek 6.1.1). V nadaljevanju prikažemo uporabo tega drugega načina.

Koraki so sledeči. Uporabnik implementira aplikacijo na razvojnem računalniku z uporabo poljubnih pripomočkov za razvoj programske opreme (ang. IDE, integrated development environment). V našem primeru smo implementirali aplikacijo Medo in ji nastavili ustrezne vrednosti parametrov za dostop do podatkovne baze MySQL (naslov IP, uporabniško ime in geslo za dostop), ki teče v unikernelu. Uporabnik nato v korenski mapi aplikacije, poimenujmo to mapo z `medo`, ustvari podmapo `meta` z dvema besedilnima datotekama, `package.yaml` in `run.yaml`. V prvo vnese metapodatke o svoji aplikaciji, podobno kot pri pripravi unikernela MySQL. Končna vsebina datoteke `package.yaml` je natanko takšna:

```
name: medo
title: Poslovna logika aplikacije Medo
author: Miha Plesko
```

V drugo datoteko, torej `run.yaml`, pa vnese podatke, ki so potrebni za zagon aplikacije. Aplikacija Medo uporablja izvajalno okolje NodeJS, za katerega ima orodje Capstan vgrajeno neposredno podporo. V omenjeni datoteki moramo zato podati le ime izvajalnega okolja in vrednosti parametrov, neposredno povezanih z aplikacijo. Končna vsebina datoteke `run.yaml` je natanko takšna:

```
runtime: node
main: /bin/server.js
env:
   PORT: 3000
```

S tem je priprava konfiguracijskih datotek zaključena. Uporabnik nato odpre ukazno okno in se z uporabo ukaza `cd` pomakne v korensko mapo aplikacije ter požene zaporedje ukazov:

```
$ capstan stack push --size 200MB
$ capstan stack run
```

Kmalu zatem je unikernel zagnan. Zadnji korak je obisk spletnega vmesnika OpenStack Horizon, kjer unikernelu dodelimo navzven dostopen naslov IP, in aplikacijo Medo lahko začnemo uporabljati.

8.3 Ovrednotenje uporabniške izkušnje po nadgradnji

Tabela 8.1 prikazuje oceno uporabniške izkušnje poganjanja testne aplikacije Medo na oblačnem ogrodju OpenStack ob uporabi nadgrajenega orodja Capstan. Postopek nastavitve orodja je popolnoma enak kot pred nadgradnjo, torej enostaven. Pri izgradnji unikernelov potrebujemo zelo malo dodatnega znanja, saj nam orodje Capstan nudi dovolj visok nivo abstrakcije. Predvsem nas ščiti pred morebitnim prevajanjem tuje izvorne kode, saj uporablja vnaprej prevedene module. Velikost in vsebina unikernela sta povsem prilagodljivi, saj se datotečni sistem pri gradnji unikernela formatira. Pri formatiranju se velikost nastavi na želeno vrednost, potem pa se nanj (poleg jedra OSv in naših lastnih aplikacijskih datotek) prenese le izbrana kombinacija modulov. Čas izgradnje unikernela se še vedno meri v sekundah, pri čemer se moramo zavedati da je odvisen od skupne velikosti in števila

Tabela 8.1: Ocena uporabniške izkušnje pri izgradnji unikernela OSv po implementaciji nadgradenj orodja Capstan.

#	METRIKA	VREDNOST
1	Kompleksnost namestitve orodij	enostavna
2	Potreba po dodatnem znanju	majhna
3	Prevajanje tuje izvorne kode	ne
4	Možnost izbire velikosti in vsebine unikernela	popolna prilagodljivost
5	Čas izgradnje unikernela	nekaj sekund
6	Integracija z ogrodjem OpenStack	podprto
7	Možnost lokalnega poganjanja unikernela	uporabno

datotek, ki jih vanj vključimo. V našem primeru smo v unikernel s podatkovno bazo MySQL vključili 295 datotek, v unikernel z aplikacijsko logiko NodeJS pa 3915 datotek. V obeh primerih je ustvarjanje unikernela trajalo sekundo ali dve. Integracija orodja Capstan z oblačnim ogrodjem OpenStack je implementirana, zato uporabniku ni potrebno ročno prenašati in zagnjati zgrajenega unikernela. Možnosti lokalnega poganjanja unikernela pri nadgradnji nismo spreminjali, torej je še vedno enako prijazna do uporabnika kot pred nadgradnjo.

Opazimo, da smo z nadgradnjo uspešno naslovili vse metrike, ki smo smo jih v tabeli 5.2 zaznali kot problematične. Pri tem so vrednosti ostalih metrik ostale nespremenjene, saj smo obdržali vse prednosti, ki so orodje odlikovale že pred nadgradnjo. Nadgrajeno orodje Capstan zato ocenjujemo kot uporabniku prijazno.

Poglavje 9

Zaključek

Tehnologija unikernel je zanimiva alternativa splošnonamenskim virtualnim strojem. Če dana aplikacija ustreza omejitvam tehnologije unikernel, bi lahko rekli, da jo je celo *priporočljivo* pognati v unikernelu, saj tako ob nižji porabi diskovnega prostora zagotovimo višjo varnost in hitrejše delovanje. Kljub temu uporaba unikernelov danes ni razširjena, saj je relativno nova in povezana z veliko kompleksnostjo priprave. Zaradi slednjega smo magistrsko delo namenili analizi trenutne stopnje kompleksnosti priprave unikernelov in jo poskusili znižati.

Obstaja množica različnih implementacij tehnologije unikernel, ki vsaka zahteva popolnoma drugačen postopek priprave. Pri tem se avtorji le redko zavedajo, da so pravzaprav implementirali novo enoto virtualizacije, nov koncept, ki ga uporabniki še niso ponotranjili. Posledično pride do problema preskoka med dvema različnima segmentoma znanja o računalniških sistemih. Avtor implementacije unikernela je zelo verjetno sistemski programer, obvladuje sistemske klice, semaforje, prekinitve in skriptni jezik Bash. Končni uporabnik tehnologije unikernel pa je spletni programer, ki na računalnik gleda kot na črno škatlo. Obvladuje namreč ogrodja za hiter razvoj spletne programske opreme in visokonivojske programske knjižnice ter jezike. Operacijski sistem je najnižji nivo abstrakcije, ki si ga končni uporabnik predstavlja. Zato ne preseneča, da skript, ki jih je pripravil avtor za pomoč pri izgradnji

unikernela, ne razume.

Tekom magistrskega dela smo se postavili v vlogo posrednika med obema segmentoma in omogočili komunikacijo med njima. Uvedli smo koncept aplikacijskih paketov, ki na nek način služi kot skupni jezik med segmentoma. Sistemski programer prevede izvorno kodo priljubljenih programov, kot sta na primer MySQL ali izvajalno oklje NodeJS, in jo naloži v javni repozitorij v obliki aplikacijskega paketa. Pri tem nima posebnih težav, saj ima znanje, potrebno za prevajanje tuje kode. Spletni programer nato uporabi te pakete, ki mu omogočijo poganjanje njegove lastne aplikacije. Pri tem se izogne vsem frustracijam, ki jih je bil po nepotrebnem deležen prej, ko je moral brez ustreznega znanja prevajati tujo izvorno kodo.

Uporaba vnaprej prevedenih aplikacijskih paketov pa ni možna pri poljubni implementaciji unikernela. Pogojena je s podporo dinamičnega povezovalnika, ki zna pravilno povezati te pakete med seboj v času izvajanja. Zato smo izmed množice tipov unikernelov izbrali ravno OSv, ki podpira omenjeno funkcionalnost. Naše nadgradnje so tako namenjene izgradnji unikernela izključno tipa OSv. Kljub temu smo prepričani, da bodo sčasoma tudi avtorji drugih implementacij unikernelov prepoznali pomembnost uporabniške izkušnje in sledili našemu zgledu priprave repozitorija vnaprej prevedenih aplikacijskih paketov ali celo uporabili pionirska paketa, ki smo ju pripravili v okviru magistrskega dela (NodeJS in MySQL).

9.1 Predlogi za nadaljnje delo

Implementirane nadgradnje orodja Capstan izboljšajo uporabniško izkušnjo izgradnje unikernela za poganjanje aplikacije na oblačnem ogrodju OpenStack, vendar je možnosti za nadaljnje delo še veliko. Ena bolj enostavnih bi bila na primer podpora za verzioniranje aplikacijskih paketov. Trenutno namreč v javnem repozitoriju lahko obstaja le ena različica aplikacijskega paketa, kar je omejujoče za uporabnike. Zahtevnejša nadgradnja bi bila na primer vgradnja orodja Capstan neposredno v oblačno ogrodje OpenStack.

Tako uporabnik ne bi več potreboval lokalne namestitve orodja, temveč bi ustrezne aplikacijske pakete izbral kar preko spletnega vmesnika OpenStack Horizon. Še korak dlje bi bila vgradnja orodja Capstan v komponento OpenStack Murano, ki ponuja katalog aplikacij. Pri tem bi združili javni repozitorij orodja Capstan s katalogom aplikacij te komponente. Možnosti za nadaljnje delo je torej še veliko.

Literatura

[1] A. F. Ackerman, L. S. Buchwald in F. H. Lewski, "Software inspections: an effective verification process", *IEEE Software*, št. 6, zv. 3, str. 31–36, May 1989.

[2] K. Adams in O. Agesen, "A comparison of software and hardware techniques for x86 virtualization", *ACM SIGOPS Operating Systems Review*, št. 40, zv. 5, str. 2–13, 2006.

[3] T. Adufu, J. Choi in Y. Kim, "Is container-based technology a winner for high performance scientific applications?" v zborniku *Network Operations and Management Symposium (APNOMS), 2015 17th Asia-Pacific*. IEEE, 2015, str. 507–510.

[4] A. Bacchelli in C. Bird, "Expectations, outcomes, and challenges of modern code review", v zborniku *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, str. 712–721.

[5] A. Bosu, J. C. Carver, C. Bird, J. Orbeck in C. Chockley, "Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft", *IEEE Transactions on Software Engineering*, št. PP, zv. 99, str. 1–1, 2016.

[6] A. Bratterud, A. A. Walla, H. Haugerud, P. E. Engelstad in K. Begnum, "Includeos: A minimal, resource efficient unikernel for cloud services", v zborniku *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, Nov 2015, str. 250–257.

[7] L. Dabbish, C. Stuart, J. Tsay in J. Herbsleb, "Social coding in github: transparency and collaboration in an open software repository", v zborniku *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*. ACM, 2012, str. 1277–1286.

[8] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian in H. Guan, "High performance network virtualization with sr-iov", *Journal of Parallel and Distributed Computing*, št. 72, zv. 11, str. 1471 – 1480, 2012, communication Architectures for Scalable Systems. Dostopno na: http://www.sciencedirect.com/science/article/pii/S0743731512000329

[9] R. Dua, A. R. Raja in D. Kakadia, "Virtualization vs containerization to support paas", v zborniku *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, March 2014, str. 610–614.

[10] P. Enberg, "A Performance Evaluation of Hypervisor, Unikernel, and Container Network I/O Virtualization", 2016.

[11] Kantee Antti, *The Design and Implementation of the Anykernel and Rump Kernels*, 2016.

[12] C. F. Kemerer in M. C. Paulk, "The impact of design and code reviews on software quality: An empirical study based on psp data", *IEEE Transactions on Software Engineering*, št. 35, zv. 4, str. 534–550, July 2009.

[13] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti in V. Zolotarov, "Osv—optimizing the operating system for virtual machines", v zborniku *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014, str. 61–72. Dostopno na: https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity

[14] P. Li, "Selecting and using virtualization solutions: our experiences with vmware and virtualbox", *Journal of Computing Sciences in Colleges*, št. 25, zv. 3, str. 11–17, 2010.

[15] Z. Li, L. OBrien, R. Ranjan in M. Zhang, "Early observations on performance of google compute engine for scientific computing", v zborniku *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, št. 1. IEEE, 2013, str. 1–8.

[16] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand in J. Crowcroft, "Unikernels: Library operating systems for the cloud", *SIGPLAN Not.*, št. 48, zv. 4, str. 461–472, Mar. 2013. Dostopno na: http://doi.acm.org/10.1145/2499368.2451167

[17] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand in J. Crowcroft, "Unikernels: Library operating systems for the cloud", *SIGPLAN Not.*, št. 48, zv. 4, str. 461–472, Mar. 2013. Dostopno na: http://doi.acm.org/10.1145/2499368.2451167

[18] S. McIntosh, Y. Kamei, B. Adams in A. E. Hassan, "The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects", v zborniku *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, str. 192–201. Dostopno na: http://doi.acm.org/10.1145/2597073.2597076

[19] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment", *Linux Journal*, št. 2014, zv. 239, str. 2, 2014.

[20] F. P. Miller, A. F. Vandome in J. McBrewster, "Amazon web services", 2010.

[21] R. Morales, S. McIntosh in F. Khomh, "Do code review practices impact design quality? a case study of the qt, vtk, and itk projects", v zborniku *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, March 2015, str. 171–180.

[22] Pavlicek Russel C., *Unikernels Beyond Containers to the Next Generation of Cloud*, 2017.

[23] K. Pepple, *Deploying openstack*. "O'Reilly Media, Inc.", 2011.

[24] G. J. Popek in R. P. Goldberg, "Formal requirements for virtualizable third generation architectures", *Communications of the ACM*, št. 17, zv. 7, str. 412–421, 1974.

[25] R. Russell, "Virtio: Towards a de-facto standard for virtual i/o devices", *SIGOPS Oper. Syst. Rev.*, št. 42, zv. 5, str. 95–103, Jul. 2008. Dostopno na: http://doi.acm.org/10.1145/1400097.1400108

[26] M. Ryer, *Go Programming Blueprints*. Packt Publishing Ltd, 2015.

[27] O. Sefraoui, M. Aissaoui in M. Eleuldj, "Openstack: toward an open-source solution for cloud computing", *International Journal of Computer Applications*, št. 55, zv. 3, 2012.

[28] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier in L. Peterson, "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors", v zborniku *ACM SIGOPS Operating Systems Review*, št. 41, zv. 3. ACM, 2007, str. 275–287.

[29] S. Tilkov in S. Vinoski, "Node.js: Using javascript to build high-performance network programs", *IEEE Internet Computing*, št. 14, zv. 6, str. 80–83, Nov 2010.

[30] J. Turnbull, *The Docker Book: Containerization is the new virtualization*. James Turnbull, 2014.

[31] B. Wilder, *Cloud architecture patterns: using microsoft azure.* "O'Reilly Media, Inc.", 2012.

[32] M. B. Zanjani, H. Kagdi in C. Bird, "Automatically recommending peer reviewers in modern code review", *IEEE Transactions on Software Engineering*, št. 42, zv. 6, str. 530–543, June 2016.